CH-2: Combinational Logic Design

Contemporary Logic Design

YONSEI UNIVERSITY

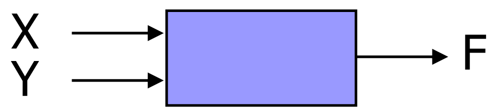
Fall 2016

Combinational Logic

- Basic logic
  - Boolean algebra, proofs by re-writing, proofs by perfect induction
  - logic functions, truth tables, and switches
  - NOT, AND, OR, NAND, NOR, XOR, . . ., minimal set
- Logic realization
  - two-level logic and canonical forms
  - incompletely specified functions
- Simplification
  - uniting theorem
  - grouping of terms in Boolean functions
- Alternate representations of Boolean functions
  - cubes
  - Karnaugh maps

Possible Logic Functions of 2-Vars

- There are *16 possible functions* for 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, <u>we can do it with only NOR or only NAND</u>
  - 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 <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))

- But learn this conversion later
  - Let’s 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		
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 • Y) + (X’ • Y’) ≡ X = Y

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

X, Y are Boolean algebra variables

Axioms/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. $(X \bullet Y) \bullet Z = X \bullet (Y \bullet Z)$

Axioms/Theorems of Boolean Algebra

- distributivity:
  8. $X \bullet (Y + Z) = (X \bullet Y) + (X \bullet Z)$ 8D. $X + (Y \bullet Z) = (X + Y) \bullet (X + Z)$
- uniting:
  9. $X \bullet Y + X \bullet Y' = X$ 9D. $(X + Y) \bullet (X + Y') = X$
- absorption:
  10. $X + X \bullet Y = X$ 10D. $X \bullet (X + Y) = X$
  11. $(X + Y') \bullet Y = X \bullet Y$ 11D. $(X \bullet Y') + Y = X + Y$
- factoring:
  12. $(X + Y) \bullet (X' + Z) = X \bullet Z + X' \bullet Y$ 12D. $X \bullet Y + X' \bullet Z = (X + Z) \bullet (X' + Y)$
- concensus:
  13. $(X \bullet Y) + (Y \bullet Z) + (X' \bullet Z) = X \bullet Y + X' \bullet Z$ 13D. $(X + Y) \bullet (Y + Z) \bullet (X' + Z) = (X + Y) \bullet (X' + Z)$

Axioms/Theorems of Boolean Algebra

- de Morgan's theorme:

  14. $(X + Y + \ldots)' = X' \bullet Y' \bullet \ldots$ 14D. $(X \bullet Y \bullet \ldots)' = X' + Y' + \ldots$

- generalized de Morgan's:

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

- For any given Boolean EQ, its inverted EQ
  - AND is changed into OR
  - OR is changed into AND
  - Each variable is changed into its inverted form

Axioms/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 true* 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 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)	$X \bullet Y + X \bullet Y'$	$=$	$X \bullet (Y + Y')$
complementarity (5)	$X \bullet (Y + Y')$	$=$	$X \bullet (1)$
identity (1D)	$X \bullet (1) = X$ ✓		

$= X$ ✓

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

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)	$X \bullet (1) = X$ ✓		

Activity

- Prove the following using the laws of Boolean algebra:
  - $(X \cdot Y) + (Y \cdot Z) + (X' \cdot Z) = X \cdot Y + X' \cdot Z$

	$(X \bullet Y) + (Y \bullet Z) + (X' \bullet Z)$
identity	$(X \bullet Y) + (1) \bullet (Y \bullet Z) + (X' \bullet Z)$
complementarity	$(X \bullet Y) + (X' + X) \bullet (Y \bullet Z) + (X' \bullet Z)$
distributivity	$(X \bullet Y) + (X' \bullet Y \bullet Z) + (X \bullet Y \bullet Z) + (X' \bullet Z)$
commutativity	$(X \bullet Y) + (X \bullet Y \bullet Z) + (X' \bullet Y \bullet Z) + (X' \bullet Z)$
factoring	$(X \bullet Y) \bullet (1 + Z) + (X' \bullet Z) \bullet (1 + Y)$
null	$(X \bullet Y) \bullet (1) + (X' \bullet Z) \bullet (1)$
identity	$(X \bullet Y) + (X' \bullet Z)$ ✓

Proving Theorems (perfect induction)

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

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

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

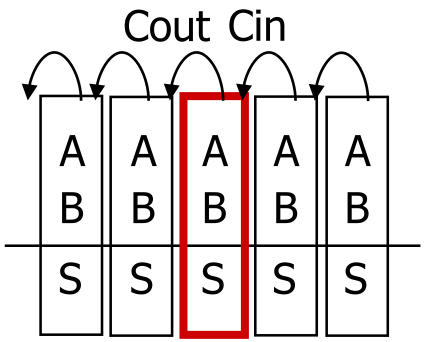
$(X \bullet Y)' = X' + Y'$

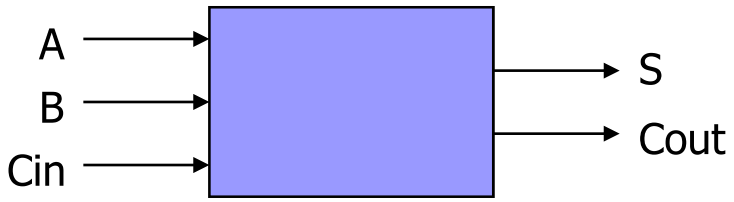
NAND is equivalent to OR with inputs complemented

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

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

Activity

- Fill in the truth-table for a circuit that checks that a 4-bit number is divisible by 2, 3, or 5

X8	X4	X2	X1	By2	By3	By5
0	0	0	0	1	1	1
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	0	1	1	0	1	0

- Write down Boolean expressions for By2, By3, and By5

Activity

X8	X4	X2	X1	By2	By3	By5
0	0	0	0	1	1	1
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	0	1	1	0	1	0
0	1	0	0	1	0	0
0	1	0	1	0	0	1
0	1	1	0	1	1	0
0	1	1	1	0	0	0
1	0	0	0	1	0	0
1	0	0	1	0	1	0
1	0	1	0	1	0	1
1	0	1	1	0	0	0
1	1	0	0	1	1	0
1	1	0	1	0	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	1

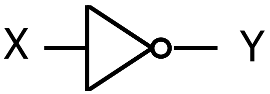
By2 = X8'X4'X2'X1' + X8'X4'X2X1'
+ X8'X4X2'X1' + X8'X4X2X1'
+ X8X4'X2'X1' + X8X4'X2X1'
+ X8X4X2'X1' + X8X4X2X1'
= X1'

By3 = X8'X4'X2'X1' + X8'X4'X2X1
+ X8'X4X2X1' + X8X4'X2'X1
+ X8X4X2'X1' + X8X4X2X1

By5 = X8'X4'X2'X1' + X8'X4X2'X1
+ X8X4'X2X1' + X8X4X2X1

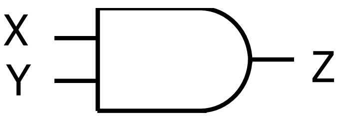
Boolean Expressions to Logic Gates

- NOT   X’ $\underline{X}$ ~X



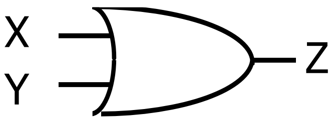
X	Y
0	1
1	0

- AND $X \bullet Y$ XY $X \wedge Y$



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

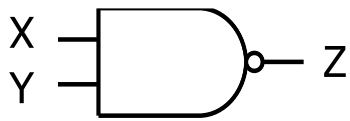
- OR $X + Y$ $X \vee Y$



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

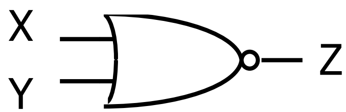
Boolean Expressions to Logic Gates

- NAND



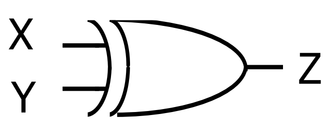
X	Y	Z
0	0	1
0	1	1
1	0	1
1	1	0

- NOR



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

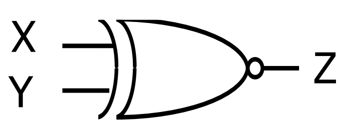
- XOR
  $X \oplus Y$



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
  $X = Y$

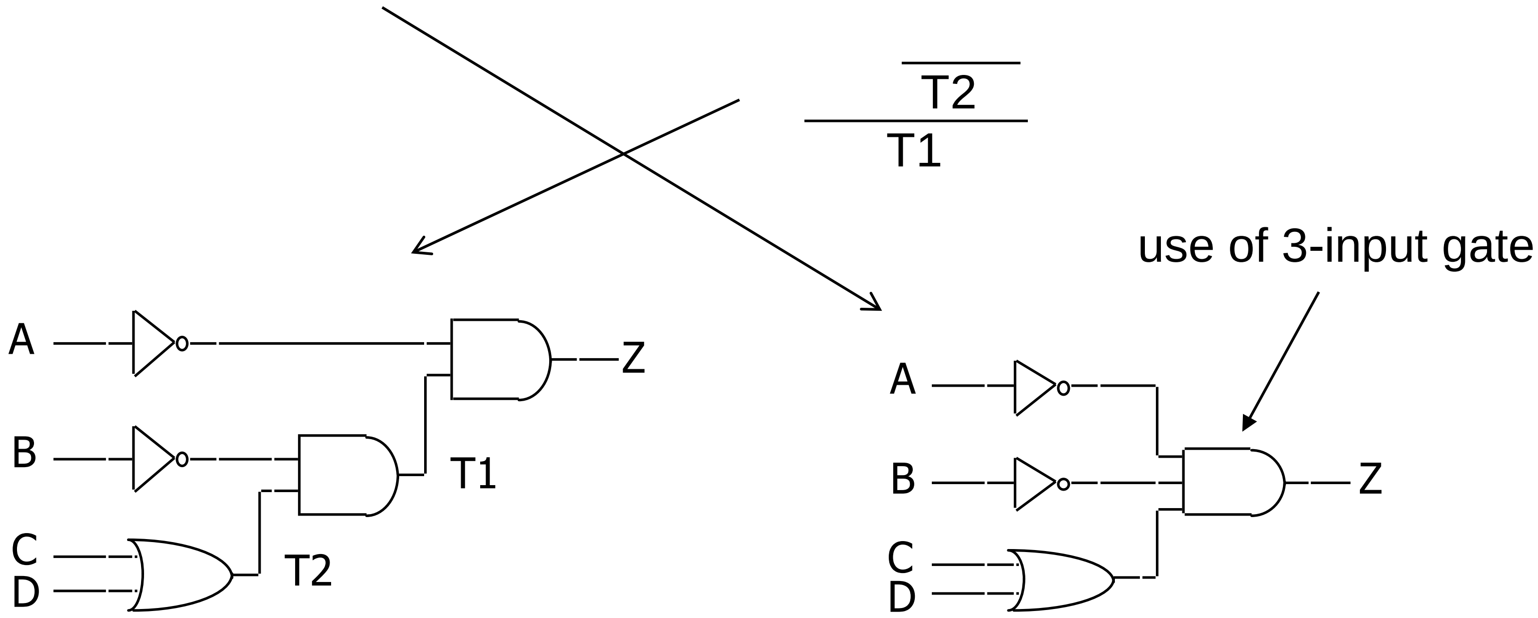


X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	1

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

Boolean Expressions to Logic Gates

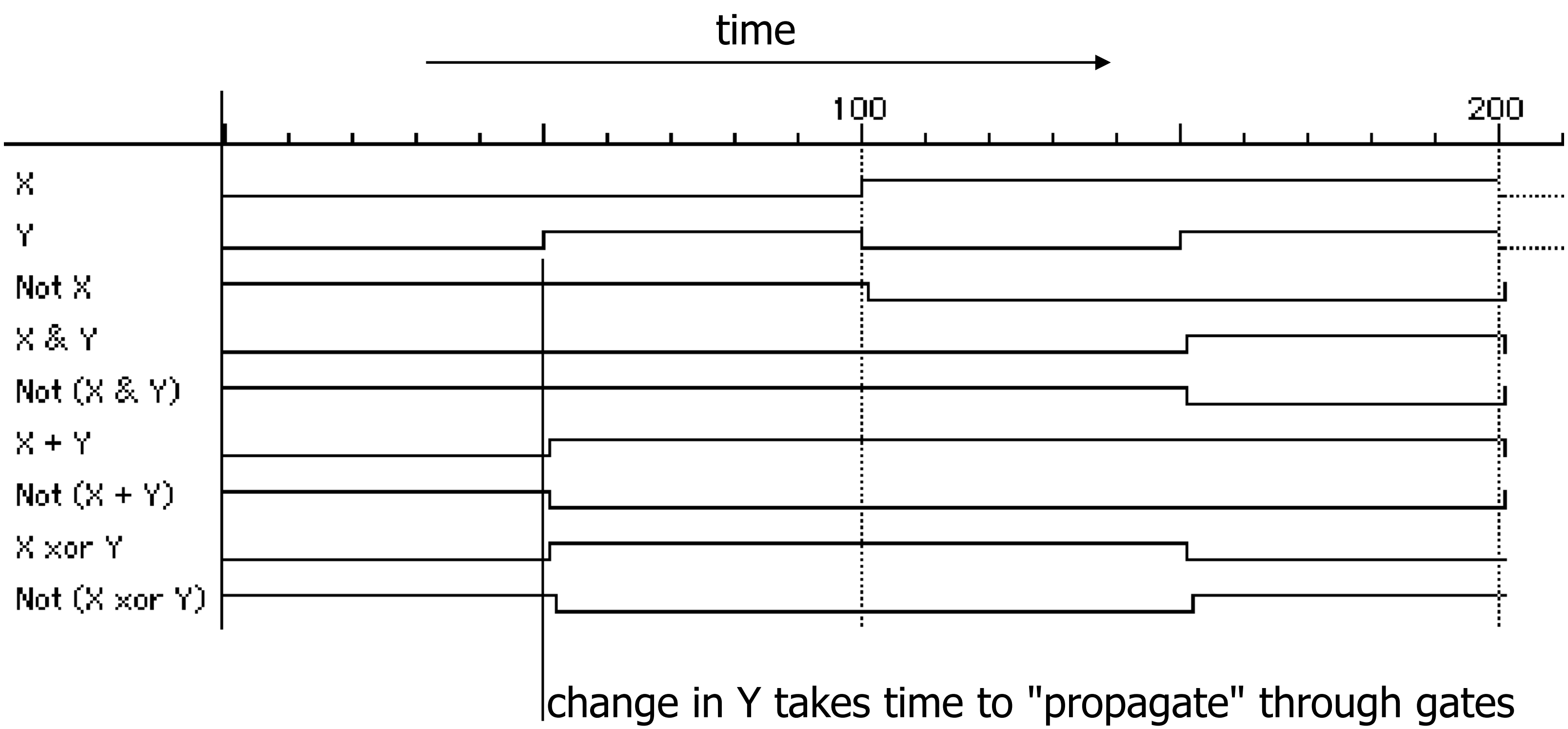
- More than one way to map expressions to gates
  - e.g., Z = A' • B' • (C + D) = (A' • (B' • (C + D)))



Literal: **each appearance of a variable or its complement in an expression**

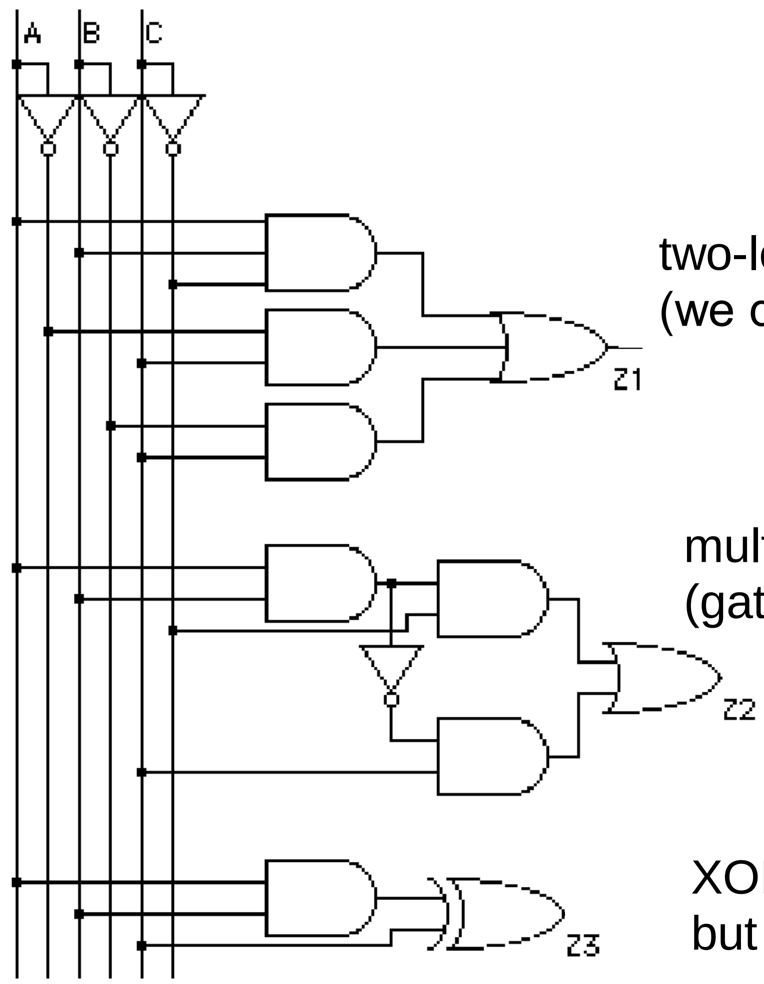
Waveform View of Logic Functions

- Just another way for truth table
  - but note how edges don't line up exactly
  - it takes time for a gate to switch its output!



Different Realizations of a Function

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



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

multi-level realization
(gates with fewer inputs)

XOR gate (easier to draw
but costlier to build)

Different Realizations of a Function

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

$Z1 = A'C(B'+B) + AB'C + ABC'$ *(5, B'+B =1, Complementarity)*

$= A'C + AB'C + ABC'$

$= C(A' + AB') + ABC'$ *(8D, Distributive law)*

$= C((A' + A)(A' + B')) + ABC'$ *(5, A'+A =1, Complementarity)*

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

$= ABC' + A'C + B'C$

$Z2 = ABC' + A'C + B'C$

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

$= ABC' + C(AB)'$ *(12, A' + B' = (AB), DeMorgan's Law)*

$Z3 = ABC' + C(AB)'$ *(X=AB, Y= C,)*

$= AB \oplus C$

Which Realization is Best?

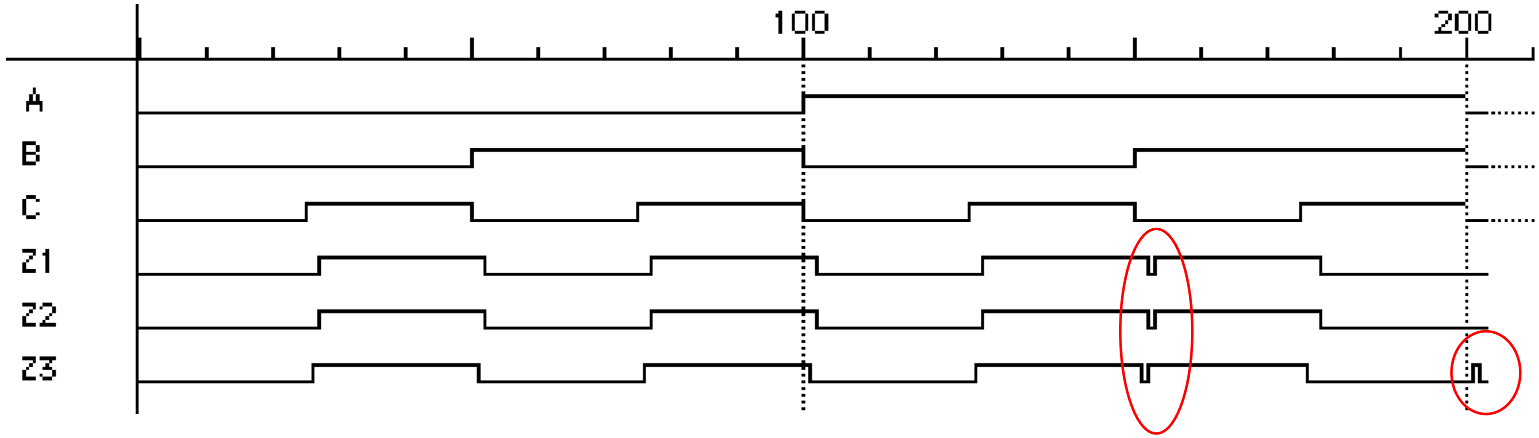
- *Reduce* number of inputs
  - literal: input variable (complemented or not)
    - can approximate the cost of logic gate as 2 transistors per literal
    - why not count inverters?
  - fewer literals means less transistors
    - smaller circuits
  - fewer inputs implies faster gates
    - gates are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

- *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: reduce while trading off against delay

Are all Realizations Equivalent?

- Under the same input stimuli, three alternative implementations have *<u>almost the same waveform behavior</u>*
  - <u>delays are different</u>
  - glitches (hazards) may arise – these could be bad, it depends
  - variations due to differences in number of gate levels and structure
- The three implementations are <u>functionally equivalent</u>



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
- 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 Canonical Forms

- Also known as disjunctive normal form
- Also known as <u>minterm expansion</u>

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

- 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	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:

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

canonical form ≠ minimal form

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

Product-of-sums Canonical Form

- Also known as conjunctive normal form
- Also known as <u>maxterm expansion</u>

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

- 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

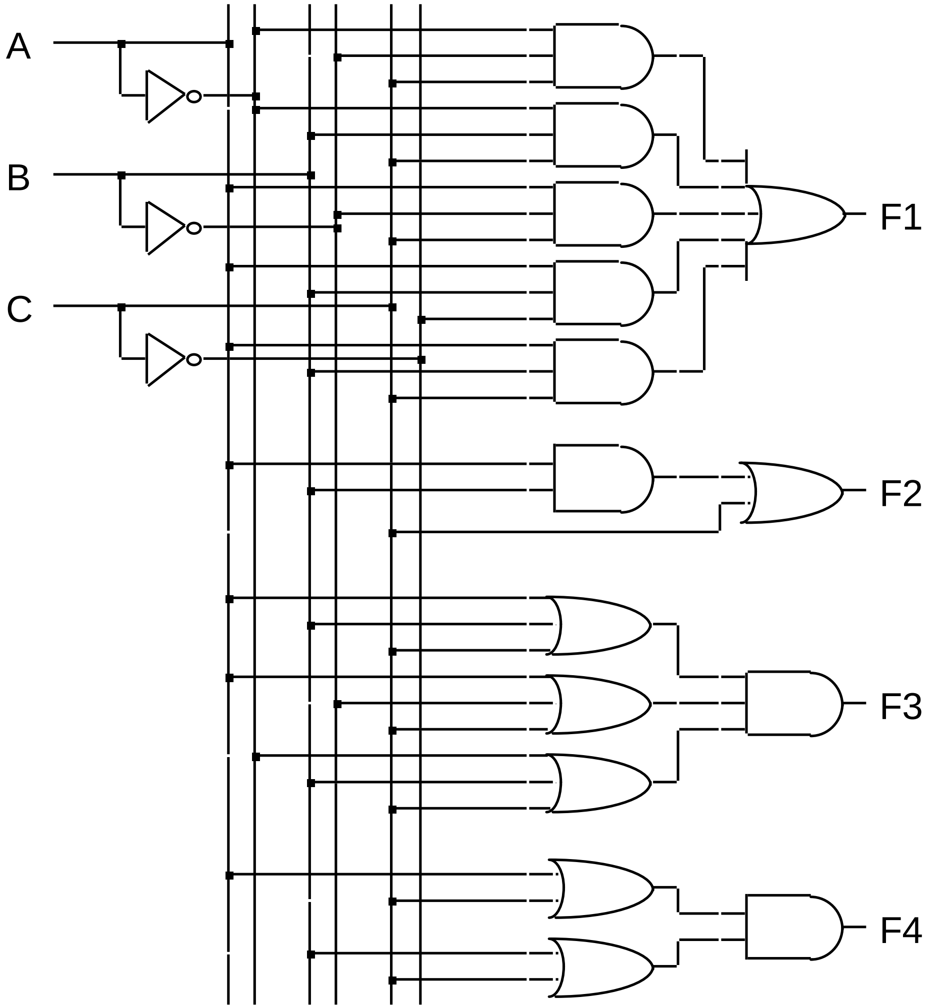
$$F(A, B, C) = (A + B + C)\,(A + B' + C)\,(A' + B + C)$$
$$= (A + B + C)\,(A + B' + C)$$
$$(A + B + C)\,(A' + B + C)$$
$$= (A + C)\,(B + C)$$

S-o-P, P-o-S, and de Morgan's Theorem

- Sum-of-products
  - F' = A'B'C' + A'BC' + AB'C' → F' in SOP
- Apply de Morgan's
  - (F')' = (A'B'C' + A'BC' + AB'C')' = (A'B'C')' • (A'BC')' •(AB'C')'
  - F = (A + B + C) (A + B' + C) (A' + B + C)
    → F in POS

- Product-of-sums → F' in POS
  - 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'))'
    = (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
    → F in SOP

Four 2-level implementations: F = AB + C



canonical sum-of-products

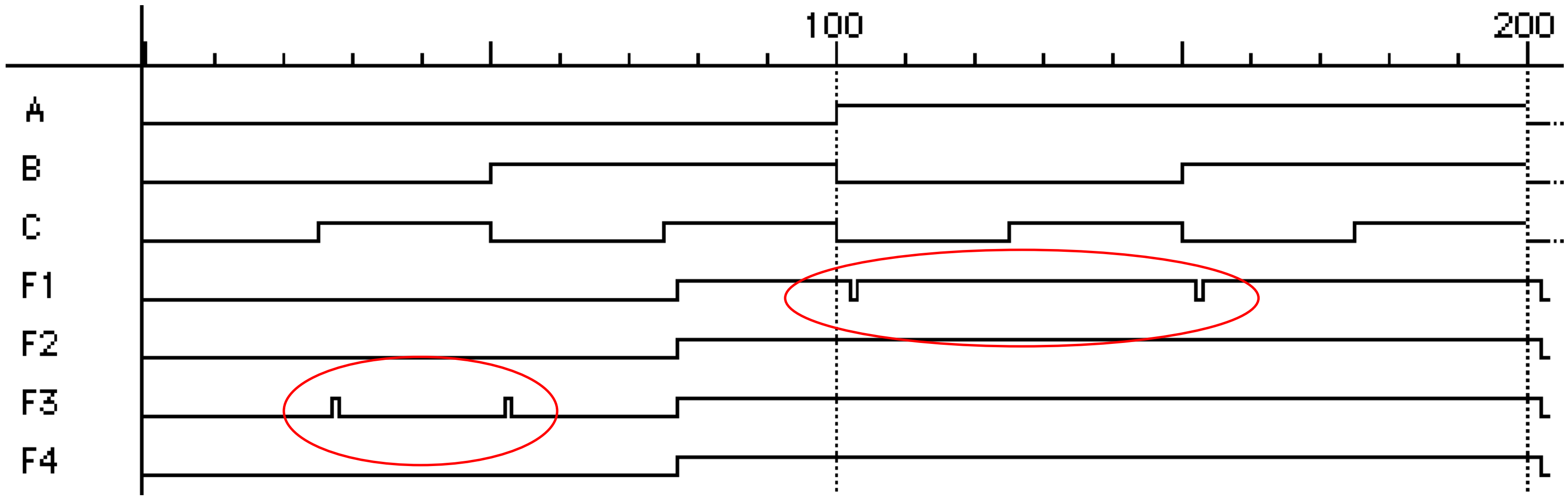
minimized sum-of-products

canonical product-of-sums

minimized product-of-sums

Waveforms for the Four Alternatives

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

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 Fts

- Don't cares and canonical forms
  - so far, only represented on-set
  - need two of the three sets (on-set, off-set, dc-set)

- Canonical representations of the BCD increment by 1 function:

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

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

Simplification of 2-level Combi-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

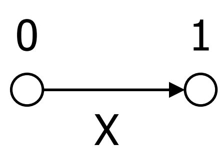
B has the same value in both on-set rows – B remains

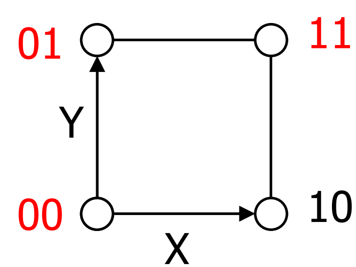
A has a different value in the two rows – A is eliminated

Boolean Cubes

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

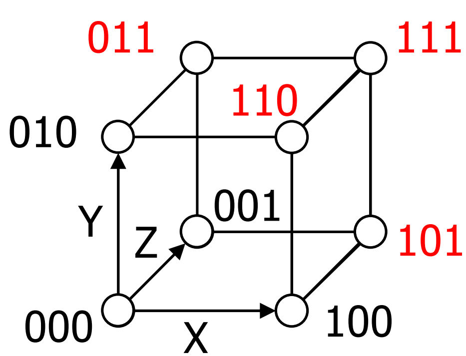
1-cube

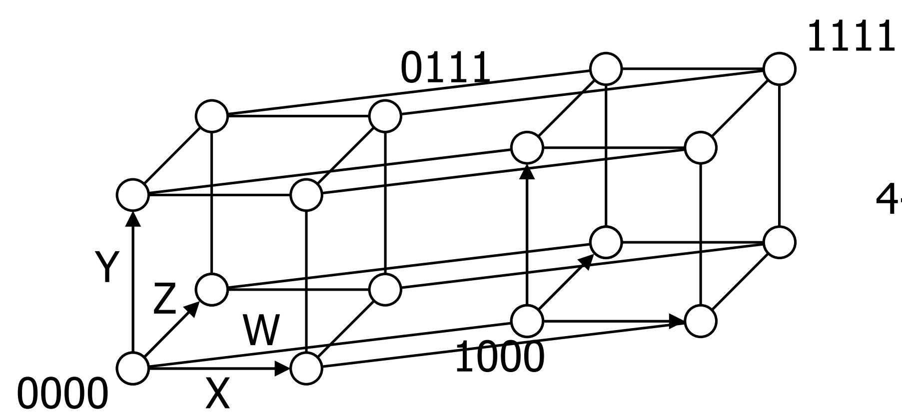




2-cube

3-cube



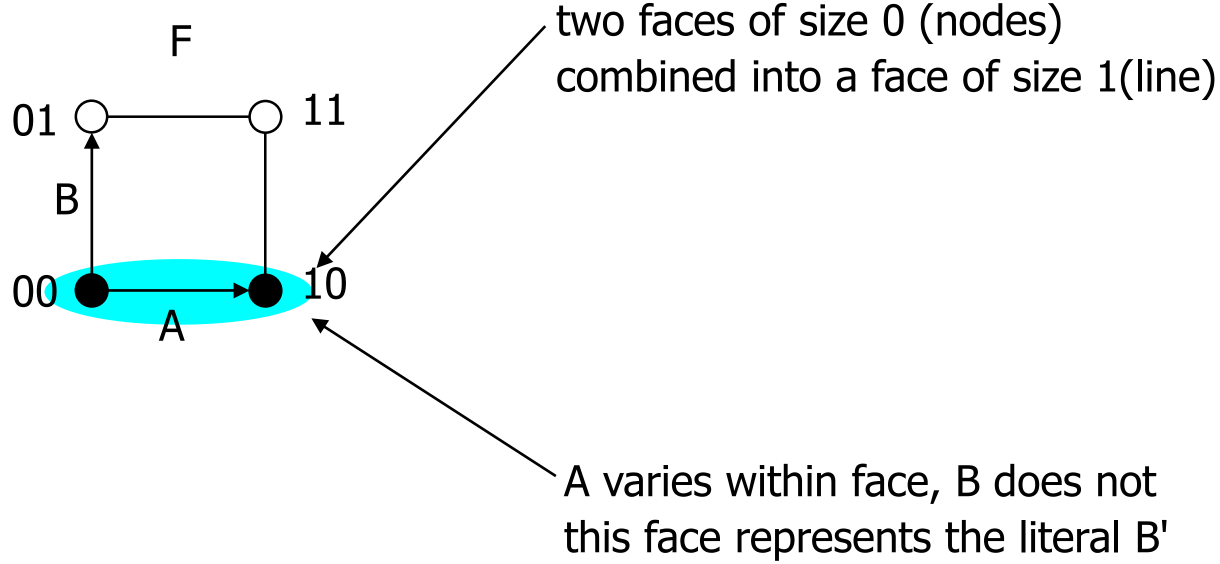


4-cube

Mapping Truth Tables to 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

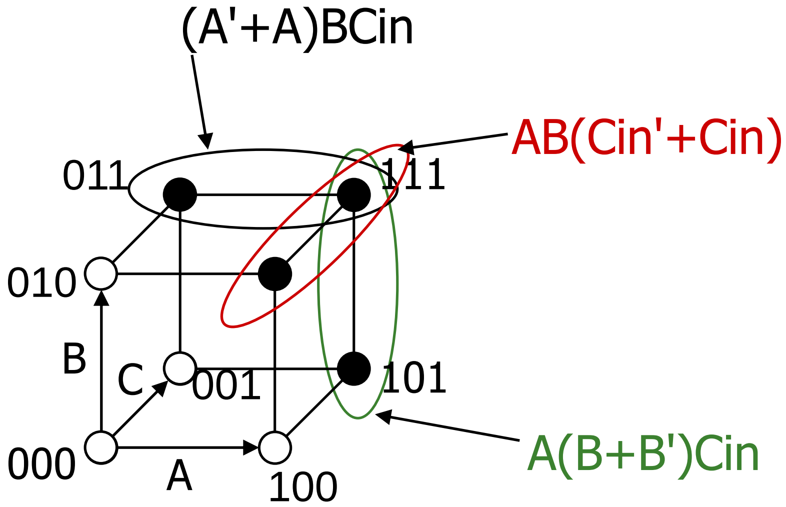


ON-set = solid nodes
OFF-set = empty nodes
DC-set = ×'d nodes

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

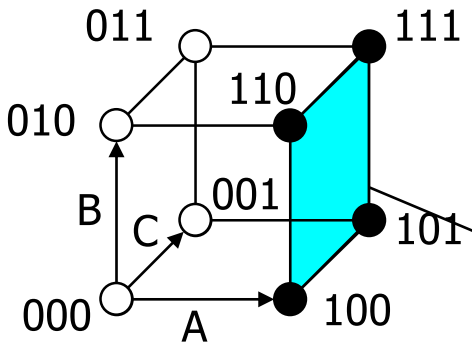


the on-set is completely covered by the combination (OR) of the subcubes of lower dimensionality - note that “111” is covered three times

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

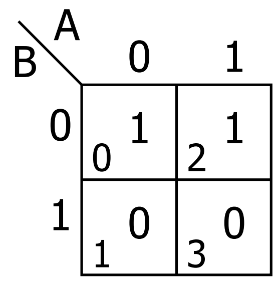
This subcube represents the literal A

m-dimensional Cubes

- In a 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

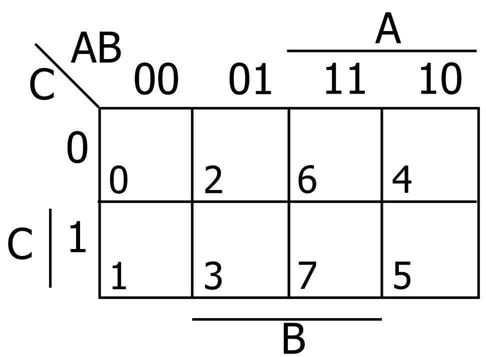
- Flat map of Boolean cube
  - wrap–around at edges
  - hard to draw and visualize for more than 4 dimensions
  - virtually impossible for more than 6 dimensions
- Alternative to truth-tables to help visualize adjacencies
  - guide to applying the uniting theorem
  - <u>on-set elements with only one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

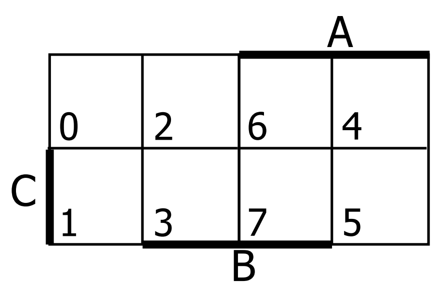


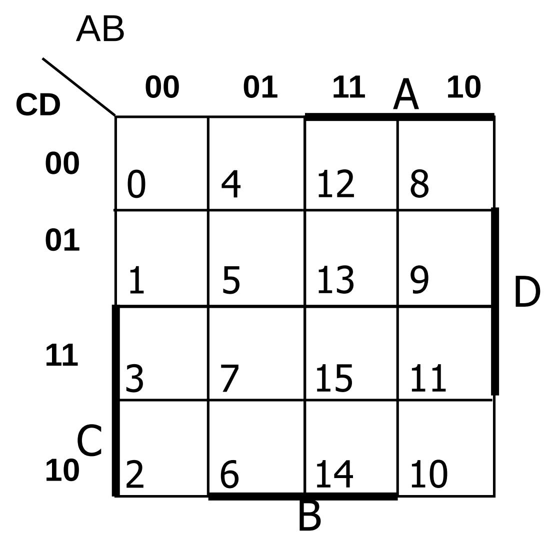
A	B	F
0	0	1
0	1	0
1	0	1
1	1	0

Karnaugh Maps

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



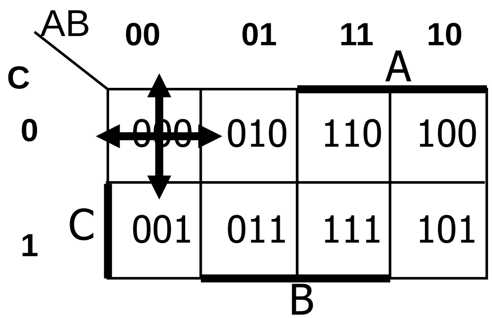


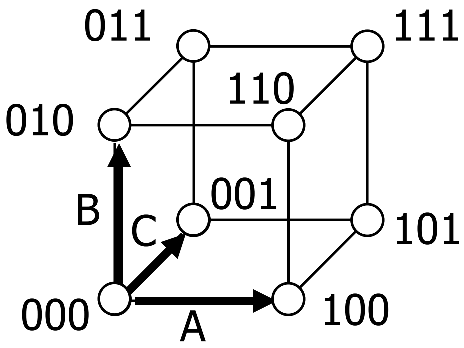


13 = 1101= ABC′D

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(0,4,5,7)

B \ A	0	1
0	1	1
1	0	0

B′

Cin \ AB	00	01	11	10
0	0	0	1	0
1	0	1	1	1

AB+ ACin + BCin

C \ AB	00	01	11	10
0	1	0	0	1
1	0	0	1	1

AC + B′C′ + ~~AB′~~

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

More Karnaugh Map Examples

C \ AB	00	01	11	10
0	0	0	1	1
1	0	0	1	1

$G(A,B,C) = A$

C \ AB	00	01	11	10
0	1	0	0	1
1	0	0	1	1

$F(A,B,C) = \Sigma m(0,4,5,7) = AC + B'C'$

C \ AB	00	01	11	10
0	0	1	1	0
1	1	1	0	0

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

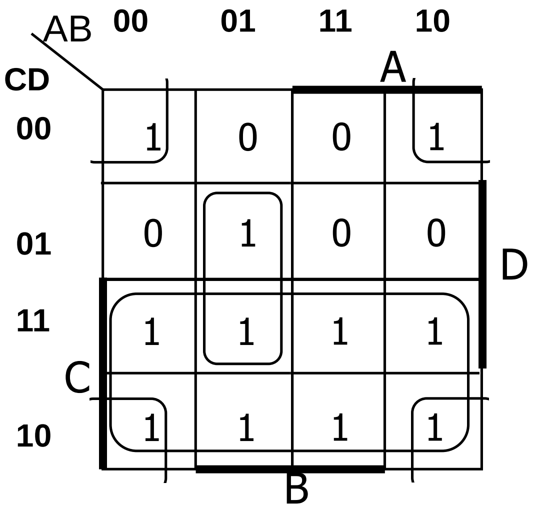
$F'(A,B,C) = \Sigma\, m(1,2,3,6) = BC' + A'C$

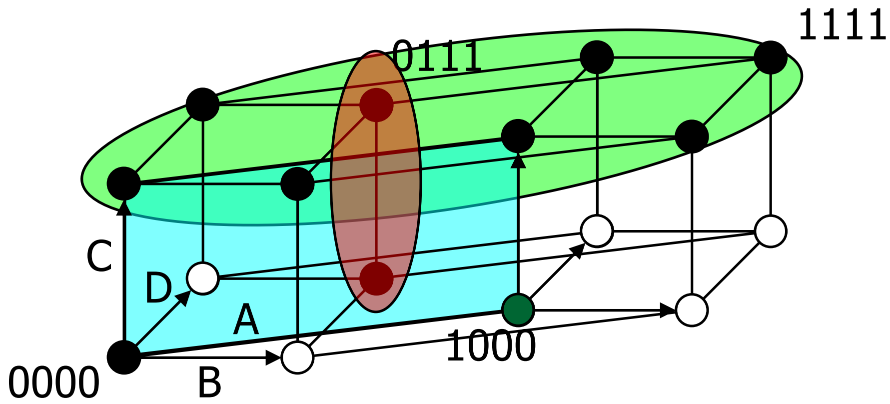
Karnaugh map: 4-variable Example

- 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

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

      A′D + B′C′D

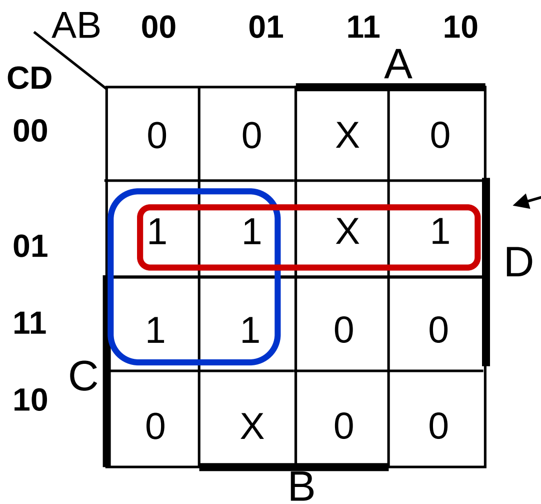
CD \ AB	00	01	11	10
00	0	0	X	0
01	1	1	X	1
11	1	1	0	0
10	0	X	0	0

A 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 = with don't cares

    A'D + C'D

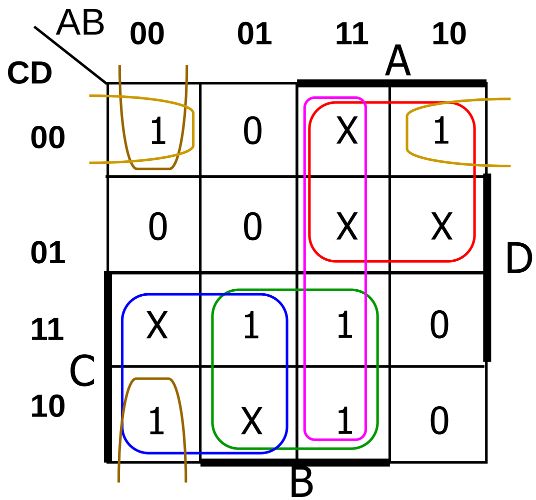


by using don't care as a "1"
a 2-cube can be formed
rather than a 1-cube to cover
this node

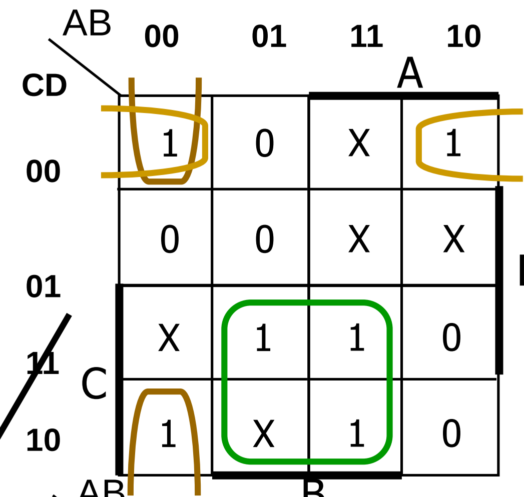
<u>don't cares</u> can be treated as
1s or 0s
depending on which is more
advantageous

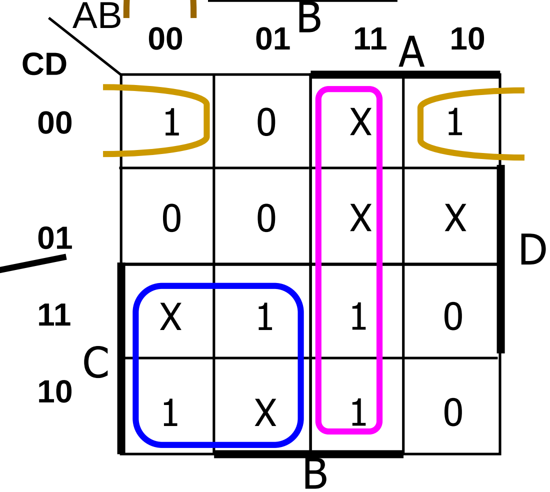
Activity

- Minimize the function F = Σ m(0, 2, 7, 8, 14, 15) + d(3, 6, 9, 12, 13)



$$F = \cancel{AC'} + \cancel{A'C} + BC + \cancel{AB} + A'B'D' + B'C'D'$$





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

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

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