# CENG-232
# Logic Design

## Lecture 2
## Combinational Logic & Circuits

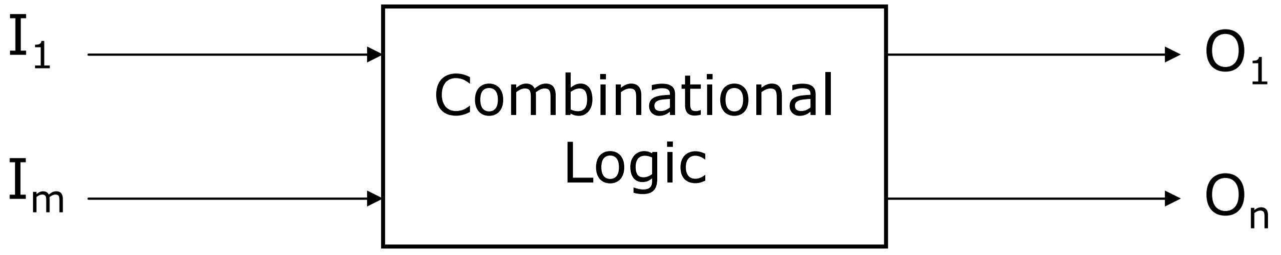
Uluç Saranlı
saranli@ceng.metu.edu.tr

## Overview

- Binary logic operations and gates
- Switching algebra
- Algebraic Minimization
- Standard forms
- Karnaugh Map Minimization
- Other logic operators
- IC families and characteristics
  - *Read Chapters 1 & 2 !*

## Combinational Logic

- One or more digital signal *inputs*
- One or more digital signal *outputs*
- Outputs are only *functions* of current input values (ideal) plus logic *propagation delay(s)*



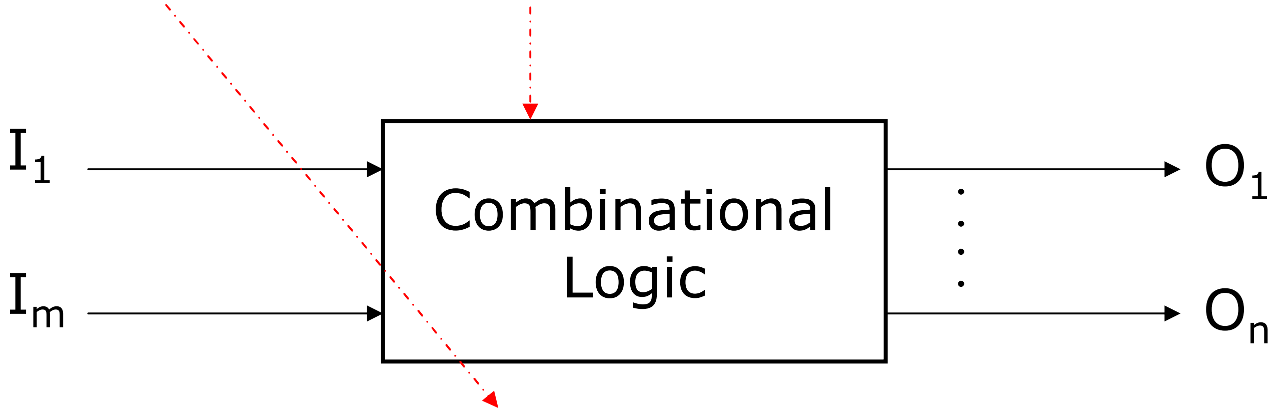
$$O_1(t+\Delta t) = F_1(I_1(t),\ldots I_m(t))$$

$$\vdots$$

$$O_n(t+\Delta t) = F_n(I_1(t),\ldots I_m(t))$$

## Engineering Parameters

- Time
  - Delay
- Space
  - # of Transistors (Chip Area)



$$O_1(t+\Delta t) = F_1\big(I_1(t), \ldots I_m(t)\big)$$

$$\vdots$$

$$O_n(t+\Delta t) = F_n\big(I_1(t), \ldots I_m(t)\big)$$

## Combinational Logic (cont.)

- Combinational logic has ***no memory!***
  - Outputs are only function of current input combination
  - Nothing is known about past events
  - Repeating a sequence of inputs always gives the same output sequence

- Sequential logic *(to be covered later)* ***has memory!*** *(over time axis; i.e., past history)*
  - Repeating a sequence of inputs can result in an entirely different output sequence

## Combinational Logic - Example

- A circuit controlling the level of fluid in a tank
  - inputs are:
    - HI - 1 if fluid level is too high, 0 otherwise
    - LO - 1 if fluid level is too low, 0 otherwise
  - outputs are:
    - Pump - 1 to pump fluid into tank, 0 for pump off
    - Drain - 1 to open tank drain, 0 for drain closed
  - input to output relationship is described by a ***truth table***

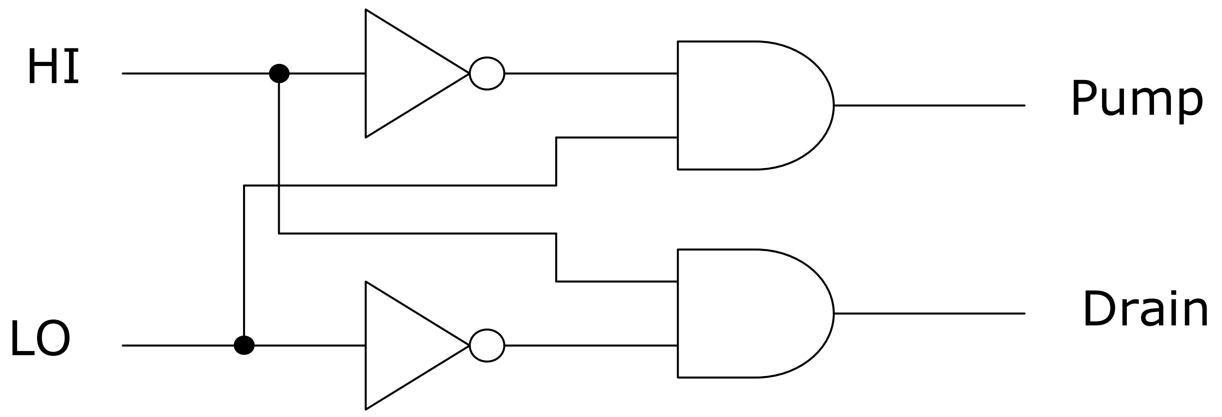
# Combinational Logic - Example

| HI | LO | Pump | Drain |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | **1** | 0 |
| 1 | 0 | 0 | **1** |
| 1 | 1 | x | x |

*Truth Table Representation*

Tank level is OK
Low level, pump more in
High level, drain some out
Inputs cannot occur

*Schematic Representation*



## Switching Algebra

- Based on Boolean Algebra
  - Developed by George Boole in 1854
  - Formal way to describe *logic statements* and determine truth of statements
- Only has two-values domain (0 and 1)
- Huntington’s Postulates define underlying assumptions

## Huntington’s Postulates

- Closure

  If X and Y are in set (0,1) then operations X+Y and $X \cdot Y$ are also in set (0,1)

- Identity Element

  $X + 0 = X$ $X \cdot 1 = X$

- Commutativity

  $X + Y = Y + X$ $X \cdot Y = Y \cdot X$

## Huntington's Postulates (cont.)

- Distributive

$$X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z)$$
$$X + (Y \cdot Z) = (X + Y) \cdot (X + Z)$$

- Complement

$$X + \overline{X} = 1$$
$$X \cdot \overline{X} = 0$$

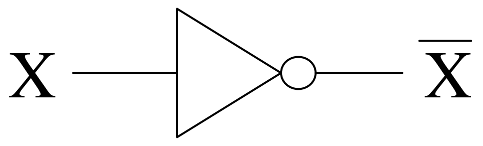
- Duality Principle:

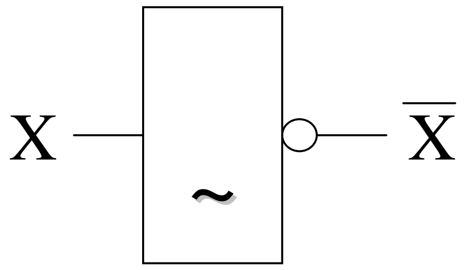
Note that for each property, one form is the dual of the other;
(0's to 1's, 1's to 0's, ·'s to +'s, +'s to ·'s)

## Switching Algebra Operations - NOT

- Unary *complement* or *inversion* operation
- Usually shown as overbar ($\overline{X}$), other forms are ~X, X′

| X | $\overline{X}$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

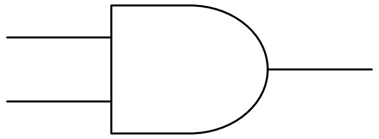


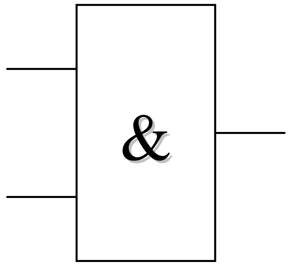


## Switching Algebra Operations - AND

- Also known as the *conjunction* operation;
  - output is true (1) only if *all* inputs are true
- Algebraic operators are "·", "&", "∧"

| X | Y | X·Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

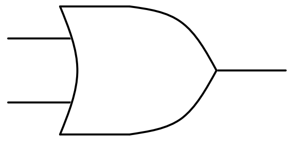


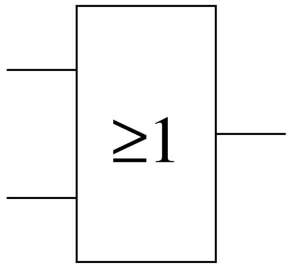


## Switching Algebra Operations - OR

- Also known as the ***disjunction*** operation;
  - output is true (1) if *any* input is true
- Algebraic operators are “+”, “|”, “$\vee$”

| X | Y | X+Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |





## Logic Expressions

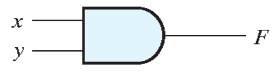
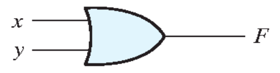
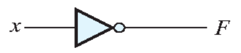
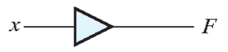
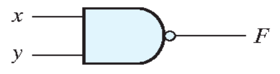
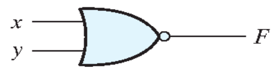
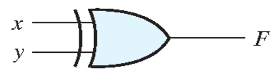
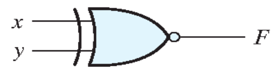
- Terms and Definitions
  - Logic Expression - a mathematical formula consisting of logical operators and variables
  - Logic Operator - a function that gives a well defined output according to switching algebra
  - Logic Variable - a symbol representing the two possible switching algebra values of 0 and 1
  - Logic Literal - the values 0 and 1 or a logic variable or it's complement

| | |
|---|---|
| $Z = X.Y + X.W$ | logic expression |
| $X, Y, W, Z$ | logic variables |
| $+, \ .$ | operators |

## Logic Expressions - Precedence

- Like standard algebra, switching algebra operators have a precedence of evaluation
  - NOT operations have the highest precedence
  - AND operations are next
  - OR operations are lowest

- Parentheses explicitly define the order of operator evaluation
  - If in doubt, use PARENTHESES!

| Name | Graphic symbol | Algebraic function | Truth table |
|---|---|---|---|
| AND |  | $F = xy$ | $x$ $y$ \| $F$<br>0 0 \| 0<br>0 1 \| 0<br>1 0 \| 0<br>1 1 \| 1 |
| OR |  | $F = x + y$ | $x$ $y$ \| $F$<br>0 0 \| 0<br>0 1 \| 1<br>1 0 \| 1<br>1 1 \| 1 |
| Inverter |  | $F = x'$ | $x$ \| $F$<br>0 \| 1<br>1 \| 0 |
| Buffer |  | $F = x$ | $x$ \| $F$<br>0 \| 0<br>1 \| 1 |
| NAND |  | $F = (xy)'$ | $x$ $y$ \| $F$<br>0 0 \| 1<br>0 1 \| 1<br>1 0 \| 1<br>1 1 \| 0 |
| NOR |  | $F = (x + y)'$ | $x$ $y$ \| $F$<br>0 0 \| 1<br>0 1 \| 0<br>1 0 \| 0<br>1 1 \| 0 |
| Exclusive-OR (XOR) |  | $F = xy' + x'y$<br>$= x \oplus y$ | $x$ $y$ \| $F$<br>0 0 \| 0<br>0 1 \| 1<br>1 0 \| 1<br>1 1 \| 0 |
| Exclusive-NOR or equivalence |  | $F = xy + x'y'$<br>$= (x \oplus y)'$ | $x$ $y$ \| $F$<br>0 0 \| 1<br>0 1 \| 0<br>1 0 \| 0<br>1 1 \| 1 |

## Logic Expressions vs. Logic Circuits

- Each expression can be realized by a logic circuit
- Multiple expressions for a single function
- Simpler the expression is, faster and cheaper the circuit is
- So we try to simplify logic expressions if possible
- We call it ‘minimization’

## Logic Expression Minimization

- Goal is to find an equivalent of an original logic expression that:
  - a) has fewer variables per term
  - b) has fewer terms
  - c) needs less logic to implement

  Example: E = A.B.C.D + A.B.C.D′ = A.B.C

- There are three main manual methods
  - Algebraic minimization
  - Karnaugh Map minimization
  - Quine-McCluskey (tabular) minimization
    - *[Assignment: Read this from the book]*

## Algebraic Minimization

- Process is to apply the switching algebra postulates, laws, and theorems(that will follow) to transform the original expression
  - Hard to recognize when a particular law can be applied
  - Difficult to know if resulting expression is truly minimal
  - Very easy to make a mistake
    - Incorrect complementation
    - Dropped variables

## Switching Algebra - Laws and Theorems

<u>Involution:</u>

$$X = \overline{\left(\overline{X}\right)}$$

## Switching Algebra - Laws and Theorems

Identity:

$$X + 1 = 1 \qquad X \cdot 0 = 0$$

$$X + 0 = X \qquad X \cdot 1 = X$$

## Switching Algebra - Laws and Theorems

Idempotence:

$$X + X = X \qquad X \cdot X = X$$

## Switching Algebra - Laws and Theorems

Associativity:

$$X + (Y + Z) = (X + Y) + Z$$

$$X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z$$

## Switching Algebra - Laws and Theorems

Adjacency:

$$X \cdot Y + X \cdot \overline{Y} = X$$

$$(X + Y) \cdot \left(X + \overline{Y}\right) = X$$

## Switching Algebra - Laws and Theorems

Absorption:

$$X + (X \cdot Y) = X$$

$$X \cdot (X + Y) = X$$

## Switching Algebra - Laws and Theorems

Simplification:

$$X + (\overline{X} \cdot Y) = X + Y$$

$$X \cdot (\overline{X} + Y) = X \cdot Y$$

## How to Prove?

- Either use the postulates and simplify

$$X+(X'.Y) = X+Y \qquad (1)$$

Use distributive postulate

$$X+YZ = (X+Y).(X+Z)$$

So (1) can be simplified as:

$$(X+X').(X+Y) = 1.(X+Y) = X+Y$$

## How to Prove?

- Truth Table Approach

| X | Y | $X+(X'.Y)$ | $X+Y$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 |

*Two output columns are the same!*

## Switching Algebra - Laws and Theorems

Consensus:

$$X \cdot Y + \overline{X} \cdot Z + Y \cdot Z = X \cdot Y + \overline{X} \cdot Z$$

$$(X + Y) \cdot (\overline{X} + Z) \cdot (Y + Z) = (X + Y) \cdot (\overline{X} + Z)$$

## Switching Algebra - Laws and Theorems

DeMorgan's Theorem:

$$\overline{X + Y} = \overline{X} \cdot \overline{Y}$$

$$\overline{X \cdot Y} = \overline{X} + \overline{Y}$$

General form:

$$\overline{F(\cdot, +, X_1, \ldots X_n)} = G(+, \cdot, \overline{X_1}, \ldots \overline{X_n})$$

## DeMorgan's Theorem

Very useful for complementing function expressions:

$e.g.$

$\mathrm{F} = \mathrm{X} + \mathrm{Y} \cdot \mathrm{Z};$ $\qquad \overline{\mathrm{F}} = \overline{\mathrm{X} + \mathrm{Y} \cdot \mathrm{Z}}$

$\overline{F} = \overline{\mathrm{X}} \cdot \overline{\mathrm{Y} \cdot \mathrm{Z}}$ $\qquad \overline{\mathrm{F}} = \overline{\mathrm{X}} \cdot \left(\overline{\mathrm{Y}} + \overline{\mathrm{Z}}\right)$

$\overline{F} = \overline{\mathrm{X}} \cdot \overline{\mathrm{Y}} + \overline{\mathrm{X}} \cdot \overline{\mathrm{Z}}$

## Minimization via Adjacency

- Adjacency is <u>easy to use</u> and <u>very powerful</u>
  - Look for two terms that are identical except for one variable

$$\text{e.g. } A \cdot B \cdot C \cdot \overline{D} \quad + \quad A \cdot B \cdot C \cdot D$$

  - Application removes one term and one variable from the remaining term

$$A \cdot B \cdot C \cdot \overline{D} \quad + \quad A \cdot B \cdot C \cdot D \quad = \quad A \cdot B \cdot C$$

$$(A \cdot B \cdot C) \cdot \overline{D} \quad + \quad (A \cdot B \cdot C) \cdot D \quad = \quad A \cdot B \cdot C$$

$$(A \cdot B \cdot C) \cdot \left(\overline{D} + D\right) \quad = \quad (A \cdot B \cdot C) \cdot 1 \quad = \quad A \cdot B \cdot C$$

# Example of Adjacency Minimization

Adjacencies

$$x_3 = \overline{b_3} b_2 \overline{b_1} b_0 + \overline{b_3} b_2 b_1 \overline{b_0} + \overline{b_3} b_2 b_1 b_0 + b_3 \overline{b_2}\, \overline{b_1}\, \overline{b_0} + b_3 \overline{b_2}\, \overline{b_1} b_0$$

Duplicate 3rd. term and rearrange (X+X=X)

$$x_3 = \overline{b_3} b_2 \overline{b_1} b_0 + \overline{b_3} b_2 b_1 b_0 + \overline{b_3} b_2 b_1 \overline{b_0} + \overline{b_3} b_2 b_1 b_0 + b_3 \overline{b_2}\, \overline{b_1}\, \overline{b_0} + b_3 \overline{b_2}\, \overline{b_1} b_0$$

Apply adjacency on "term pairs"

$$x_3 = \overline{b_3} b_2 b_0 + \overline{b_3} b_2 b_1 + b_3 \overline{b_2}\, \overline{b_1}$$

## Another Algebraic Simplification Example

W $= X'Y'Z+X'Y'Z'+XYZ+X'YZ+X'YZ'$

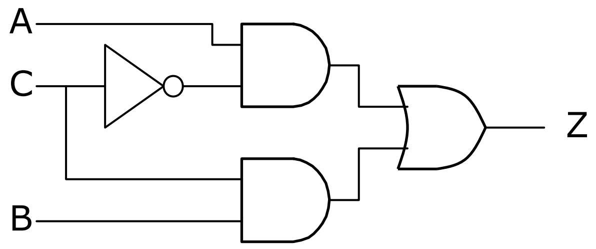
$= X'Y'+YZ+X'Z'$

*How?*

Now we study logic circuits closely and show some other easier simplification method(s).

## Combinational Circuit Analysis

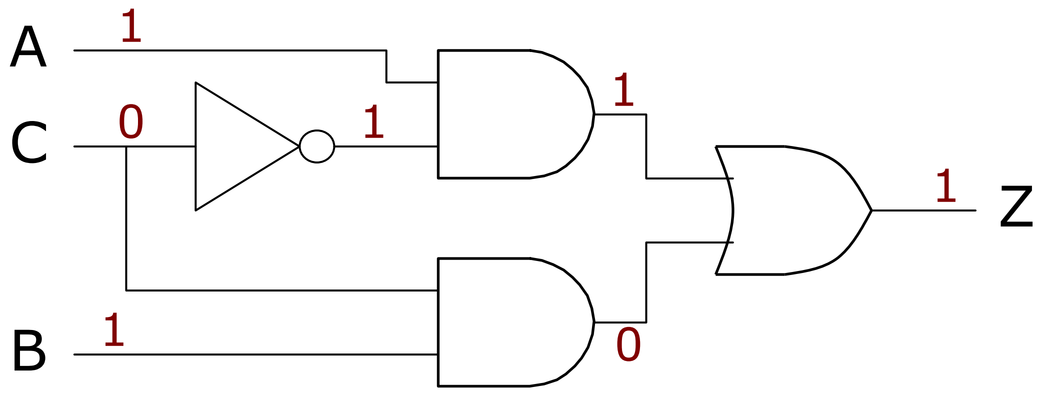
- Combinational circuit analysis starts with a <u>schematic</u> and answers the following questions:
  - What is the <u>truth table(s)</u> for the circuit output function(s)?
  - What is the <u>logic expression(s)</u> for the circuit output function(s)?



## Literal Analysis

- Literal analysis is the process of manually assigning a set of values to the inputs, tracing the results, and recording the output values
  - For “n” inputs there are 2n possible input combinations
  - From input values, gate outputs are evaluated to form next set of gate inputs
  - Evaluation continues until gate outputs are circuit outputs
- Literal analysis only gives us the truth table

## Literal Analysis - Example



| A | B | C | Z |
|---|---|---|---|
| 0 | 0 | 0 | x |
| 0 | 0 | 1 | x |
| 0 | 1 | 0 | x |
| 0 | 1 | 1 | x |
| 1 | 0 | 0 | x |
| 1 | 0 | 1 | x |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | x |

Assign input values

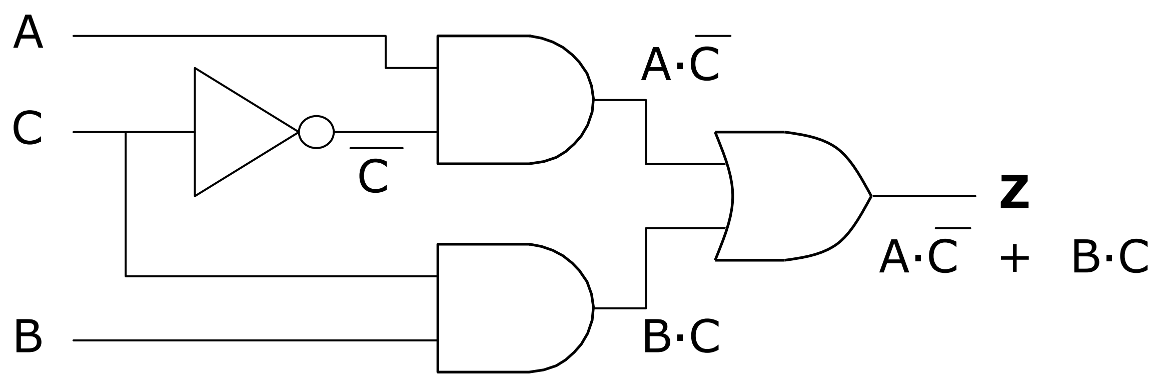
Determine gate outputs and propagate

Repeat until we reach output

## Symbolic Analysis

- Like literal analysis we start with the circuit diagram
  - Instead of assigning values, we determine gate output expressions instead
  - Intermediate expressions are combined in following gates to form complex expressions
  - We repeat until we have the output function and expression
- Symbolic analysis gives both the <u>truth table</u> and <u>logic expression</u>

## Symbolic Analysis - Example



Generate intermediate expression

Create associated TT column

Repeat till output reached

| A | B | C | $\overline{C}$ | $A\cdot\overline{C}$ | $B\cdot C$ | $Z = A\cdot\overline{C} + B\cdot C$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 |

## Symbolic Analysis (cont.)

- Note that we are constructing the truth table as we go
  - truth table has a column for each intermediate gate output
  - intermediate outputs are combined in the truth table to generate the complex columns
- Symbolic analysis is <u>more work</u> but gives us <u>complete information</u>

## Standard Expression Forms

- What if we have only the truth table?
- Can we obtain simplified expressions from the truth table?
- We need it when we do design instead of analysis
- *Consider the Liquid Tank example*

| HI | LO | Pump | Drain |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | x | x |

*By observing the pump column, can we write an expression for pump?*

***Pump = HI'.LO***

## Standard Expression Forms

- Two standard (canonical) expression forms
  - Canonical sum form
    - a.k.a., “disjunctive normal form” or *“sum-of-products”*
    - OR of AND terms
  - Canonical product form
    - a.k.a., “conjunctive normal form” or *“product-of-sums”*
    - AND of OR terms
- In both forms, each 1st-level operator corresponds to one row of truth table
- 2nd-level operator combines 1st-level results

## Standard Forms (cont.)

Standard Sum Form

Sum of Products (OR of AND terms)

$$F[A, B, C] = \left(\overline{A} \cdot \overline{B} \cdot \overline{C}\right) + \left(\overline{A} \cdot B \cdot C\right) + \left(A \cdot B \cdot \overline{C}\right) + \left(A \cdot B \cdot C\right)$$

Minterms

Standard Product Form

Product of Sums (AND of OR terms)

$$F[A, B, C] = \left(A + B + \overline{C}\right) \cdot \left(A + \overline{B} + C\right) \cdot \left(\overline{A} + B + C\right) \cdot \left(\overline{A} + B + \overline{C}\right)$$

Maxterms

## Standard Sum Form

- Each product (AND) term is a “Minterm”
  - “AND”ed product of literals in which each variable appears exactly once, in true or complemented form (but not both!)
  - Each minterm has exactly one ‘1’ in the truth table
  - When minterms are ORed together each minterm contributes a ‘1’ to the final function

NOTE:

Not all product terms are minterms!

## Minterms and Standard Sum Form

| A | B | C | Minterms | $m_0$ | $m_3$ | $m_6$ | $m_7$ | F |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | $m_0 = \overline{A}\cdot\overline{B}\cdot\overline{C}$ | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | $m_1 = \overline{A}\cdot\overline{B}\cdot C$ | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | $m_2 = \overline{A}\cdot B\cdot\overline{C}$ | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | $m_3 = \overline{A}\cdot B\cdot C$ | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | $m_4 = A\cdot\overline{B}\cdot\overline{C}$ | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | $m_5 = A\cdot\overline{B}\cdot C$ | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | $m_6 = A\cdot B\cdot\overline{C}$ | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | $m_7 = A\cdot B\cdot C$ | 0 | 0 | 0 | 1 | 1 |

$$F = \overline{A}\cdot\overline{B}\cdot\overline{C} + \overline{A}\cdot B\cdot C + A\cdot B\cdot\overline{C} + A\cdot B\cdot C$$

$$F(A,\ B,\ C) = m_0 + m_3 + m_6 + m_7$$

$$F(A,\ B,\ C) = \sum m(0,\ 3,\ 6,\ 7)$$

## Standard Product Form

- Each OR (sum) term is a “Maxterm”
  - ORed product of literals in which each variable appears exactly once, in true or complemented form (but not both!)
  - Each maxterm has exactly one ‘0’ in the truth table
  - When maxterms are ANDed together each maxterm contributes a ‘0’ to the final function

NOTE:

Not all sum terms are maxterms!

## Maxterms and Standard Product Form

| A | B | C | Maxterms | $M_1$ | $M_2$ | $M_4$ | $M_5$ | F |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | $M_0 = A+B+C$ | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | $M_1 = A+B+\overline{C}$ | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | $M_2 = A+\overline{B}+C$ | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | $M_3 = A+\overline{B}+\overline{C}$ | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | $M_4 = \overline{A}+B+C$ | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | $M_5 = \overline{A}+B+\overline{C}$ | 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | $M_6 = \overline{A}+\overline{B}+C$ | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | $M_7 = \overline{A}+\overline{B}+\overline{C}$ | 1 | 1 | 1 | 1 | 1 |

$$F = \left(A+B+\overline{C}\right)\cdot\left(A+\overline{B}+C\right)\cdot\left(\overline{A}+B+C\right)\cdot\left(\overline{A}+B+\overline{C}\right)$$

$$F\ (A,B,C) = M_1 \cdot M_2 \cdot M_4 \cdot M_5$$

$$F\ (A,B,C) = \prod M\ (1,\ 2,\ 4,\ 5)$$

## Karnaugh Map Minimization

- Given a truth table ...
- Karnaugh Map (or K-map) minimization is a ***visual minimization technique***
  - It is an application of *adjacency*
  - Procedure guarantees a minimal expression
  - Easy to use; fast
  - Problems include: *(Read this once more by the end of the chapter)*
    - Applicable to limited number of variables (4 ~ 8)
    - Errors in translation from TT to K-map
    - Not grouping cells correctly
    - Errors in reading final expression

## K-map Minimization (cont.)

- Basic K-map is a *2-D rectangular array* of cells
  - Each K-map represents one bit column of output
  - Each cell contains one bit of output function
- Arrangement of cells in array facilitates recognition of adjacent terms
  - Adjacent terms differ in one variable value; equivalent to difference of one bit of input row values
    - e.g. $m_6$ (110) and $m_7$ (111)

# Truth Table Rows and Adjacency

Standard TT ordering doesn't show adjacency

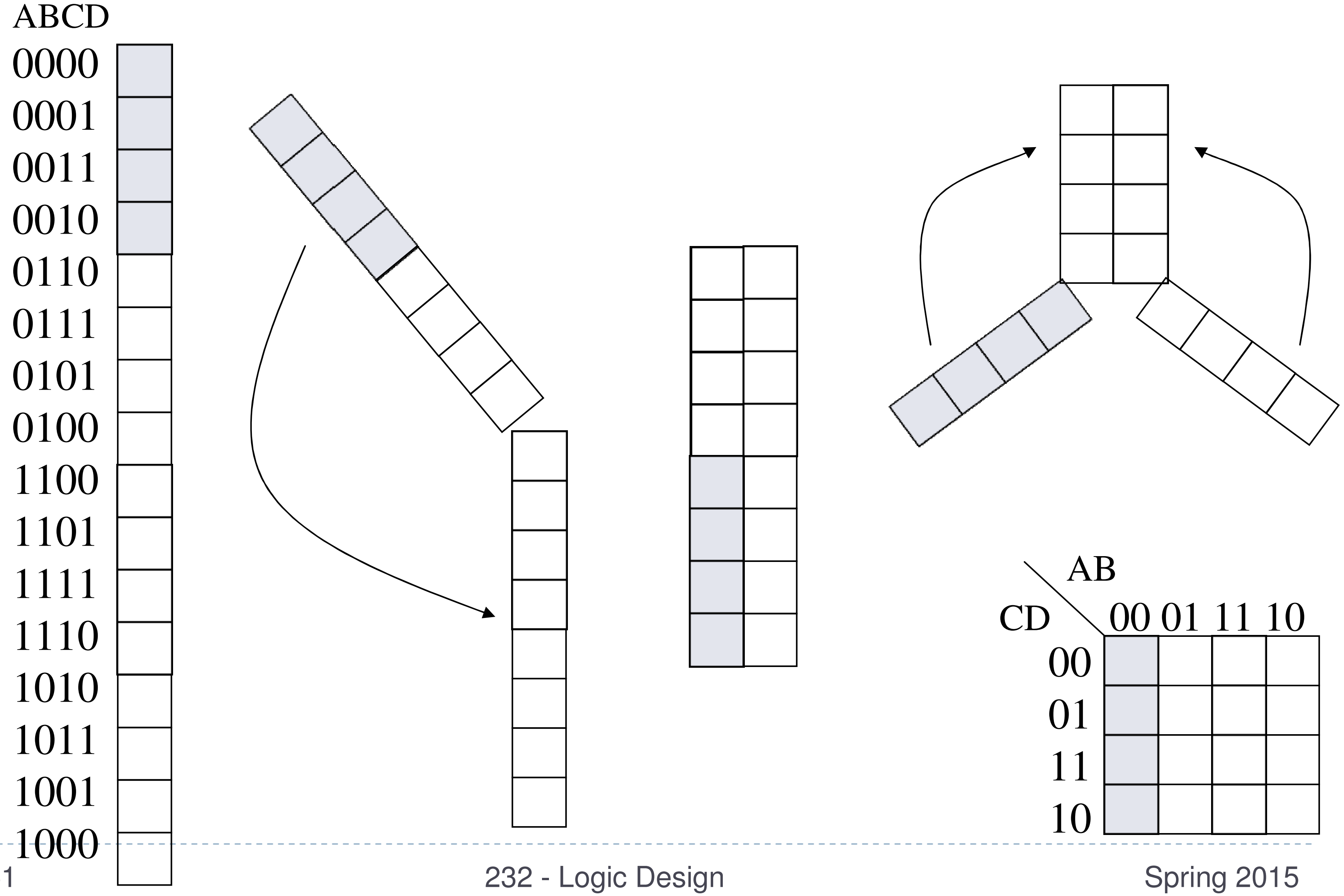
| A | B | C | D | minterm |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | $m_0$ |
| 0 | 0 | 0 | 1 | $m_1$ |
| 0 | 0 | 1 | 0 | $m_2$ |
| 0 | 0 | 1 | 1 | $m_3$ |
| 0 | 1 | 0 | 0 | $m_4$ |
| 0 | 1 | 0 | 1 | $m_5$ |
| 0 | 1 | 1 | 0 | $m_6$ |
| 0 | 1 | 1 | 1 | $m_7$ |
| 1 | 0 | 0 | 0 | $m_8$ |
| 1 | 0 | 0 | 1 | $m_9$ |
| 1 | 0 | 1 | 0 | $m_{10}$ |
| 1 | 0 | 1 | 1 | $m_{11}$ |
| 1 | 1 | 0 | 0 | $m_{12}$ |
| 1 | 1 | 0 | 1 | $m_{13}$ |
| 1 | 1 | 1 | 0 | $m_{14}$ |
| 1 | 1 | 1 | 1 | $m_{15}$ |

Key is to use gray code for row order

| A | B | C | D | minterm |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | $m_0$ |
| 0 | 0 | 0 | 1 | $m_1$ |
| 0 | 0 | 1 | 1 | $m_3$ |
| 0 | 0 | 1 | 0 | $m_2$ |
| 0 | 1 | 1 | 0 | $m_6$ |
| 0 | 1 | 1 | 1 | $m_7$ |
| 0 | 1 | 0 | 1 | $m_5$ |
| 0 | 1 | 0 | 0 | $m_4$ |
| 1 | 1 | 0 | 0 | $m_{12}$ |
| 1 | 1 | 0 | 1 | $m_{13}$ |
| 1 | 1 | 1 | 1 | $m_{15}$ |
| 1 | 1 | 1 | 0 | $m_{14}$ |
| 1 | 0 | 1 | 0 | $m_{10}$ |
| 1 | 0 | 1 | 1 | $m_{11}$ |
| 1 | 0 | 0 | 1 | $m_9$ |
| 1 | 0 | 0 | 0 | $m_8$ |

*This helps but it's still hard to see all possible adjacencies.*

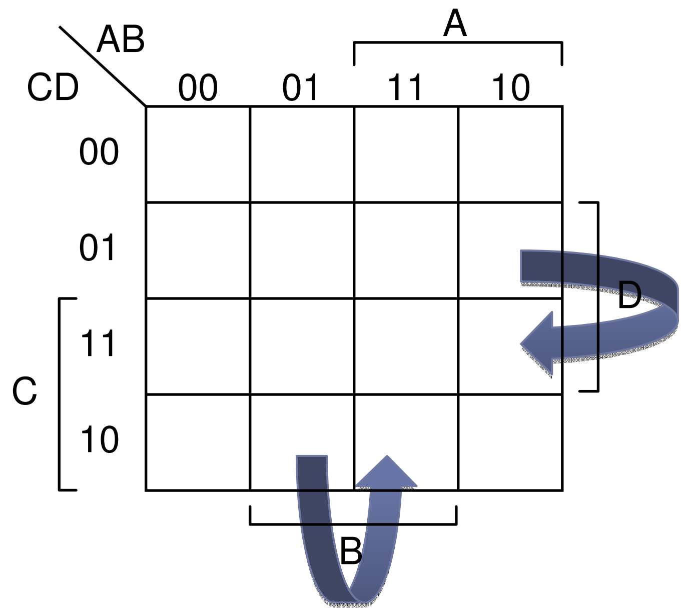
# Folding of Gray Code Table into K-map



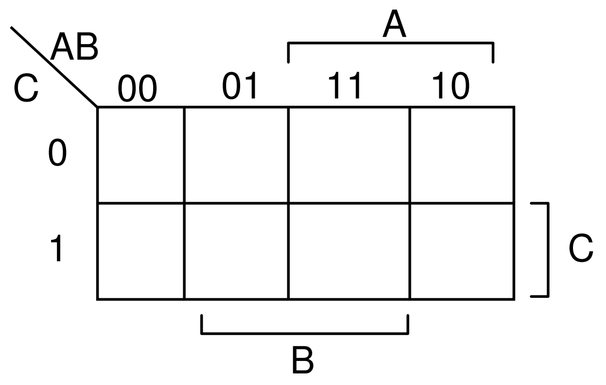
## K-map Minimization (cont.)

- For any cell in 2-D array, there are four direct neighbors (top, bottom, left, right)
- 2-D array can therefore show adjacencies of up to four variables.

Four variable K-map



Three variable K-map

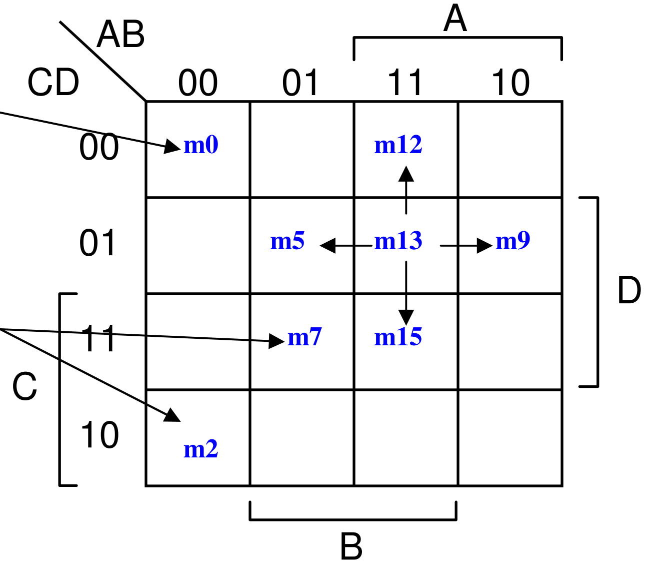


Don’t forget that cells are adjacent <u>top to bottom</u> and <u>side to side</u>. It also <u>wraps around!</u>

## Truth Table to K-map

*Number of TT rows MUST match number of K-map cells*

| A | B | C | D | F |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | x |
| 0 | 0 | 0 | 1 | x |
| 0 | 0 | 1 | 0 | x |
| 0 | 0 | 1 | 1 | x |
| 0 | 1 | 0 | 0 | x |
| 0 | 1 | 0 | 1 | x |
| 0 | 1 | 1 | 0 | x |
| 0 | 1 | 1 | 1 | x |
| 1 | 0 | 0 | 0 | x |
| 1 | 0 | 0 | 1 | x |
| 1 | 0 | 1 | 0 | x |
| 1 | 0 | 1 | 1 | x |
| 1 | 1 | 0 | 0 | x |
| 1 | 1 | 0 | 1 | x |
| 1 | 1 | 1 | 0 | x |
| 1 | 1 | 1 | 1 | x |



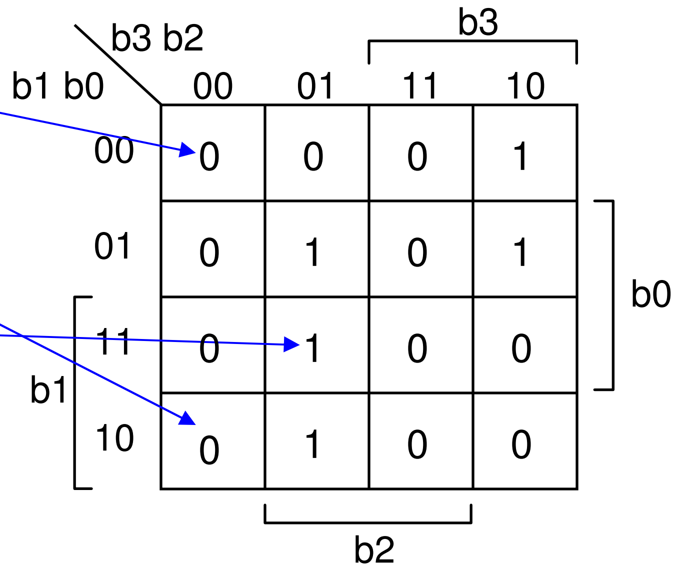
*Note different ways K-map is labeled*

# K-Map Minimization Example

Entry of TT data into K-map

| b3 | b2 | b1 | b0 | x3 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | - |
| 1 | 0 | 1 | 1 | - |
| 1 | 1 | 0 | 0 | - |
| 1 | 1 | 0 | 1 | - |
| 1 | 1 | 1 | 0 | - |
| 1 | 1 | 1 | 1 | - |

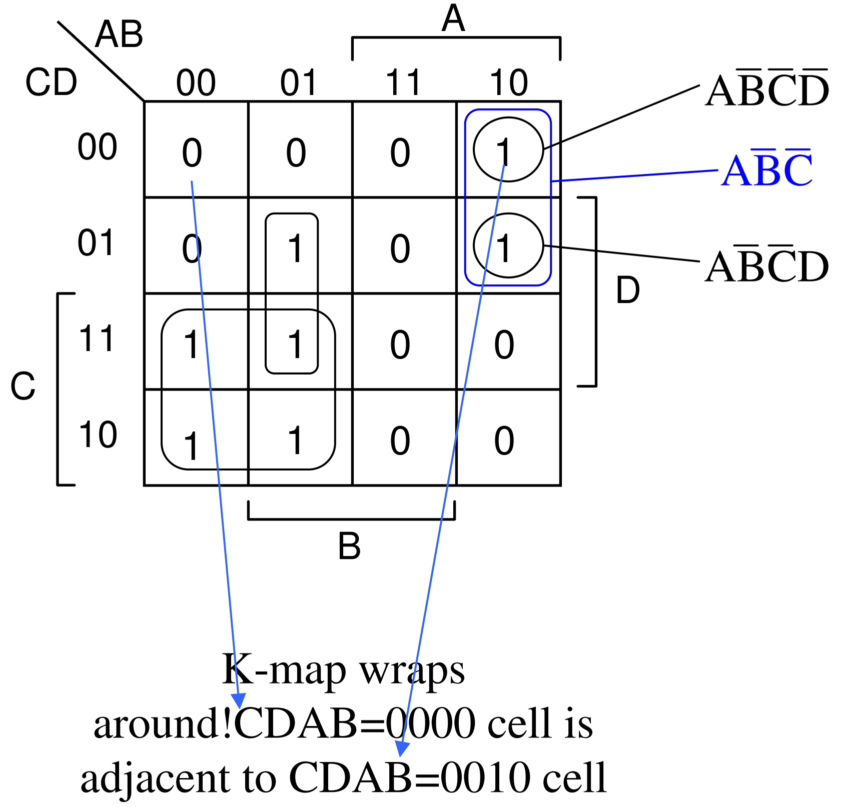
Use 0's for now



| b1 b0 \ b3 b2 | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 1 |
| 01 | 0 | 1 | 0 | 1 |
| 11 | 0 | 1 | 0 | 0 |
| 10 | 0 | 1 | 0 | 0 |

*Watch out for ordering of 10 and 11 rows and columns!*

## Grouping - Applying Adjacency



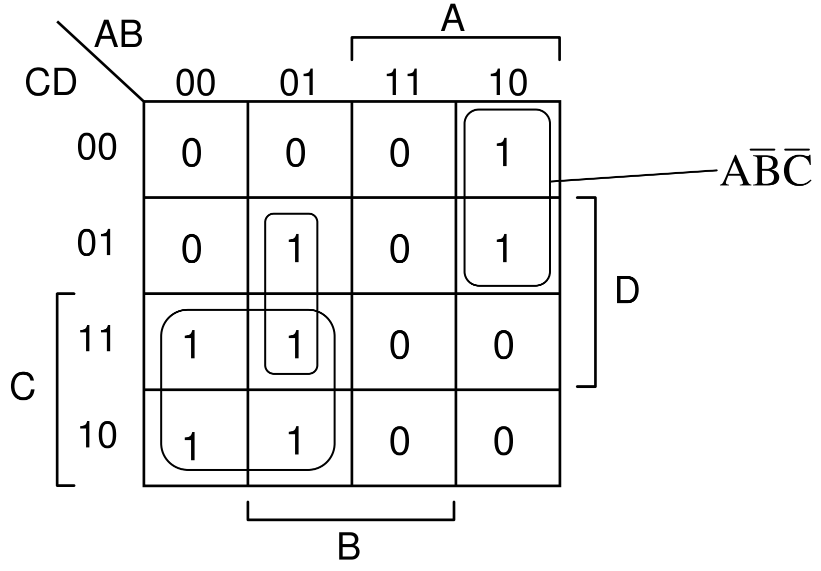
If two cells have the same value (1 here) and are next to each other, the terms are adjacent. This adjacency is shown by enclosing them.

Groups can have common cells.

Group size is a power of 2 and groups are rectangular (1,2,4,8 rectangular cells).

You can group 0s or 1s.

## Reading the Groups



If 1's are grouped, the expression is a product term, 0s - sum term.

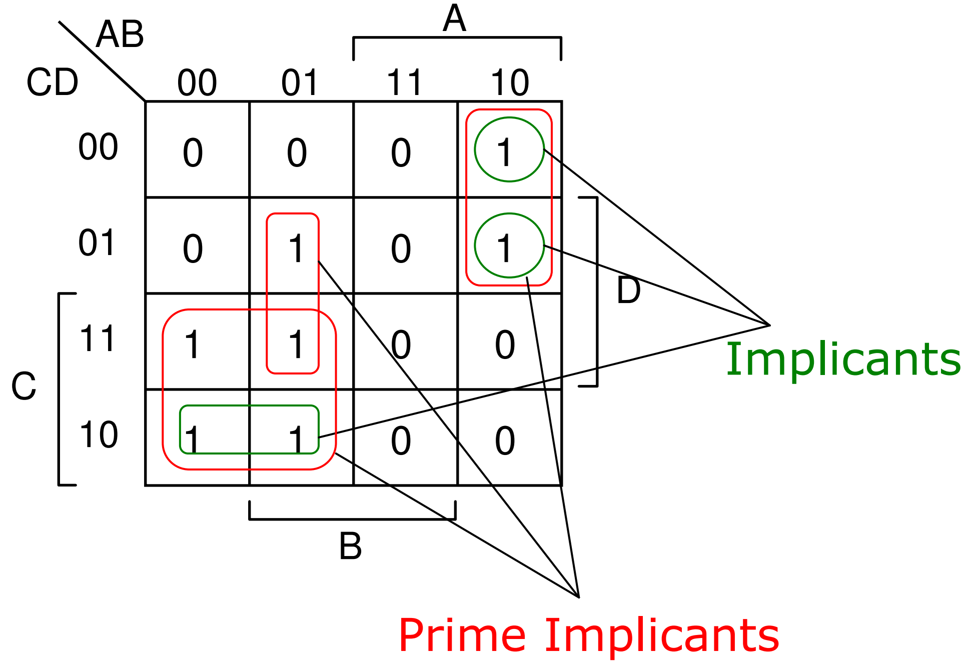
Within a group, note when variable values change as you go cell to cell. This determines how the term expression is formed by the following table

| | Grouping 1's | Grouping 0's |
|---|---|---|
| Variable changes | Exclude | Exclude |
| Variable constant 0 | Inc. comp. | Inc. true |
| Variable constant 1 | Inc. true | Inc. comp. |

## Reading the Groups (cont.)

- When reading the term expression…
  - If the associated variable value changes within the group, the variable is *dropped* from the term
  - If reading 1's, a constant 1 value indicates that the associated variable is *true in the AND* term
  - If reading 0's, a constant 0 value indicates that the associated variable is *true in the OR* term

# Implicants and Prime Implicants



Single cells or groups that could be part of a larger group are known as implicants

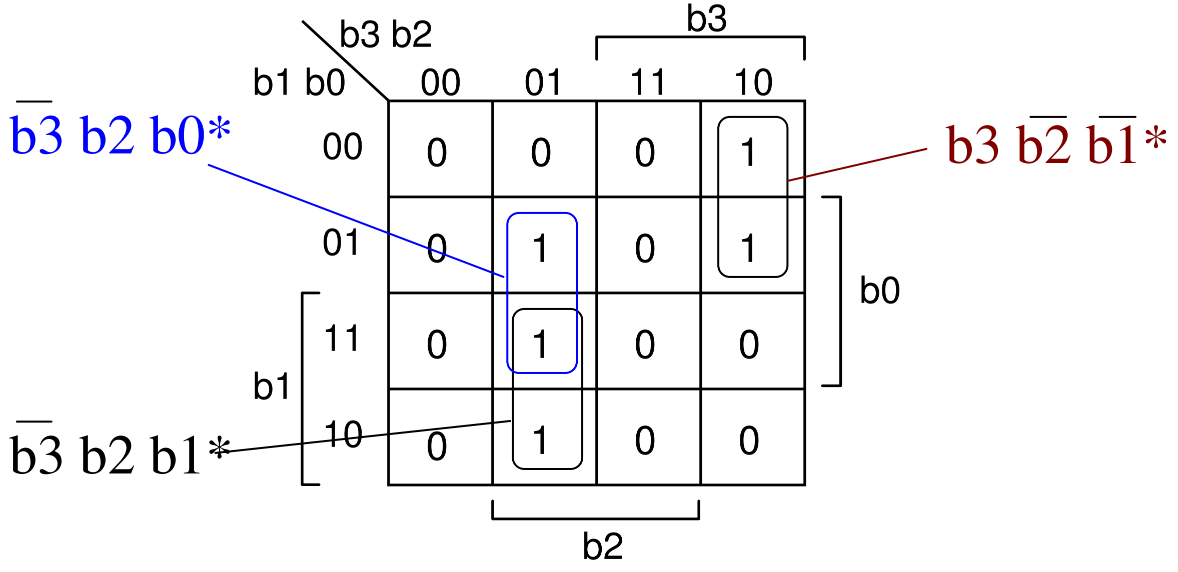
A group that is as large as possible is a prime implicant (i.e., it cannot be grouped by other implicants to eliminate a variable)

Single cells can be prime implicants if they cannot be grouped with any other cell

## Implicants and Minimal Expressions

- Any set of implicants that encloses (covers) all values is *“sufficient”*;
  - i.e. the associated logical expression represents the desired function.
  - All minterms or maxterms are sufficient.
- The smallest set of prime implicants that covers all values forms a minimal expression for the desired function.
  - There may be *more than one* minimal set.

## K-map Minimization - Example



| b1 b0 \ b3 b2 | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 1 |
| 01 | 0 | 1 | 0 | 1 |
| 11 | 0 | 1 | 0 | 0 |
| 10 | 0 | 1 | 0 | 0 |

We want a sum of products expression so we circle 1s.

* PIs are essential; no implicants remain (no secondary PIs).

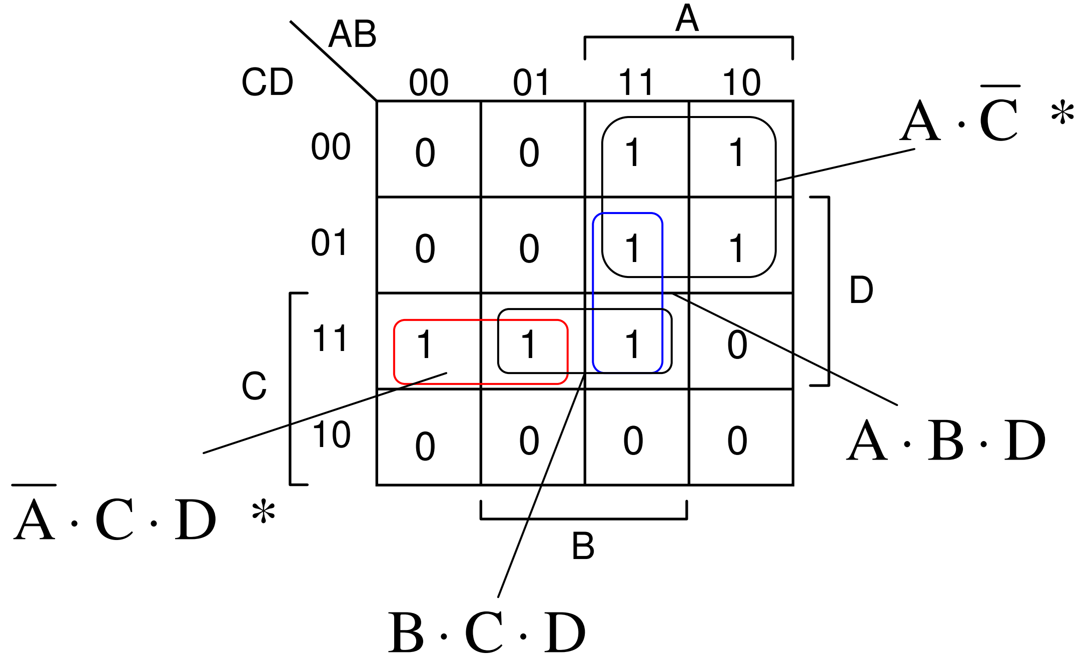
The minimal expression is:

$$X_3 = b3\ \overline{b2}\ \overline{b1} + \overline{b3}\ b2\ b1 + \overline{b3}\ b2\ b0$$

## Essential and Secondary Prime Implicants

- If a prime implicant has any cell that is not covered by any other prime implicant, it is an "*essential prime implicant*"
- If a prime implicant is not essential is is a "*secondary* prime implicant" or "*nonessential*" P.I.
- A minimal set includes *ALL essential prime implicants and the minimum number of secondary PIs* as needed to cover all values.

## K-map Minimization - Example



| CD \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 1 | 1 |
| 01 | 0 | 0 | 1 | 1 |
| 11 | 1 | 1 | 1 | 0 |
| 10 | 0 | 0 | 0 | 0 |

We want a sum of products expression so we circle 1s.

(*) PIs are essential; and we have 2 secondary PIs.

The minimal expressions are:

$$F = A \cdot \overline{C} + \overline{A} \cdot C \cdot D + B \cdot C \cdot D$$

$$F = A \cdot \overline{C} + \overline{A} \cdot C \cdot D + A \cdot B \cdot D$$

## K-map Minimization Method

- Technique is valid for either 1’s or 0’s

Find all prime implicants (largest groups of 1’s or 0’s in order of largest to smallest)

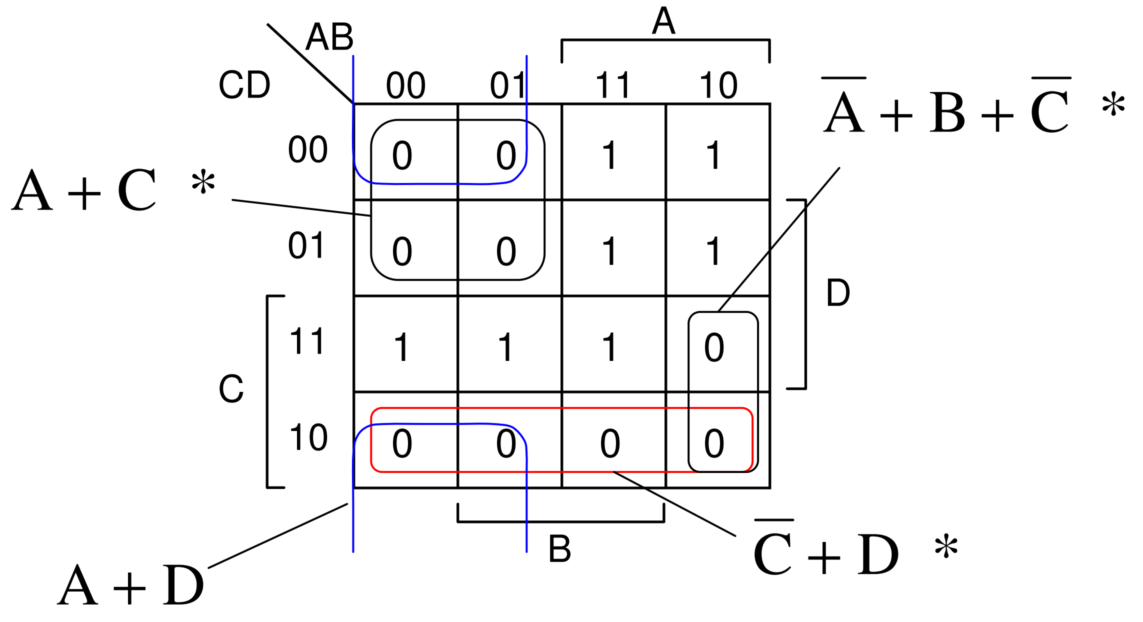
Identify minimal set of PIs

1) Find all essential PIs

2) Find smallest set of secondary PIs

The resulting expression is *minimal*.

# A 3rd K-map Minimization Example



| CD \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 1 | 1 |
| 01 | 0 | 0 | 1 | 1 |
| 11 | 1 | 1 | 1 | 0 |
| 10 | 0 | 0 | 0 | 0 |

We want a product of sums expression so we circle 0s.

(*) PIs are essential; and we have 1 secondary PI which is redundant.

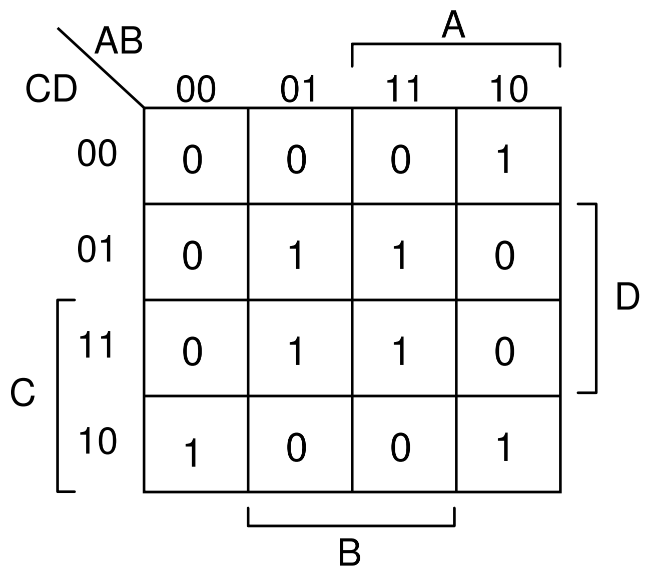
The minimal expression is:

$$F = (A + C) \cdot (\overline{C} + D) \cdot (\overline{A} + B + \overline{C})$$

*!!! Study this …*

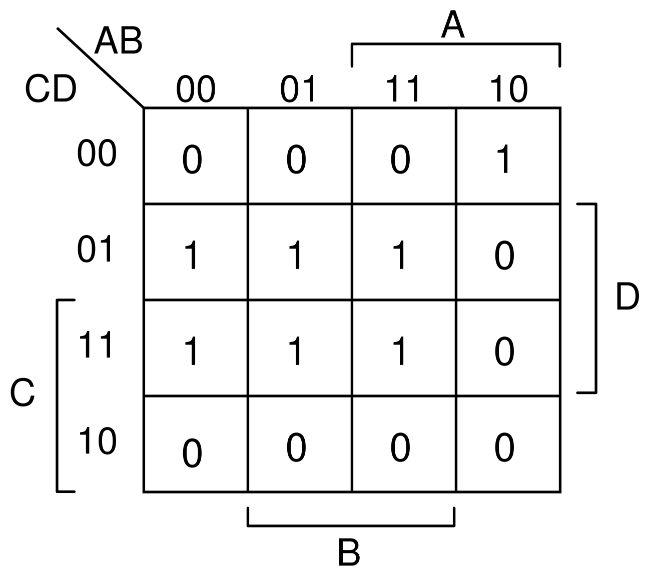
## 5 Variable K-Maps

- Uses two 4 variable maps side-by-side
  - groups spanning both maps occupy the same place in both maps



| CD \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 1 |
| 01 | 0 | 1 | 1 | 0 |
| 11 | 0 | 1 | 1 | 0 |
| 10 | 1 | 0 | 0 | 1 |

**E = 0**

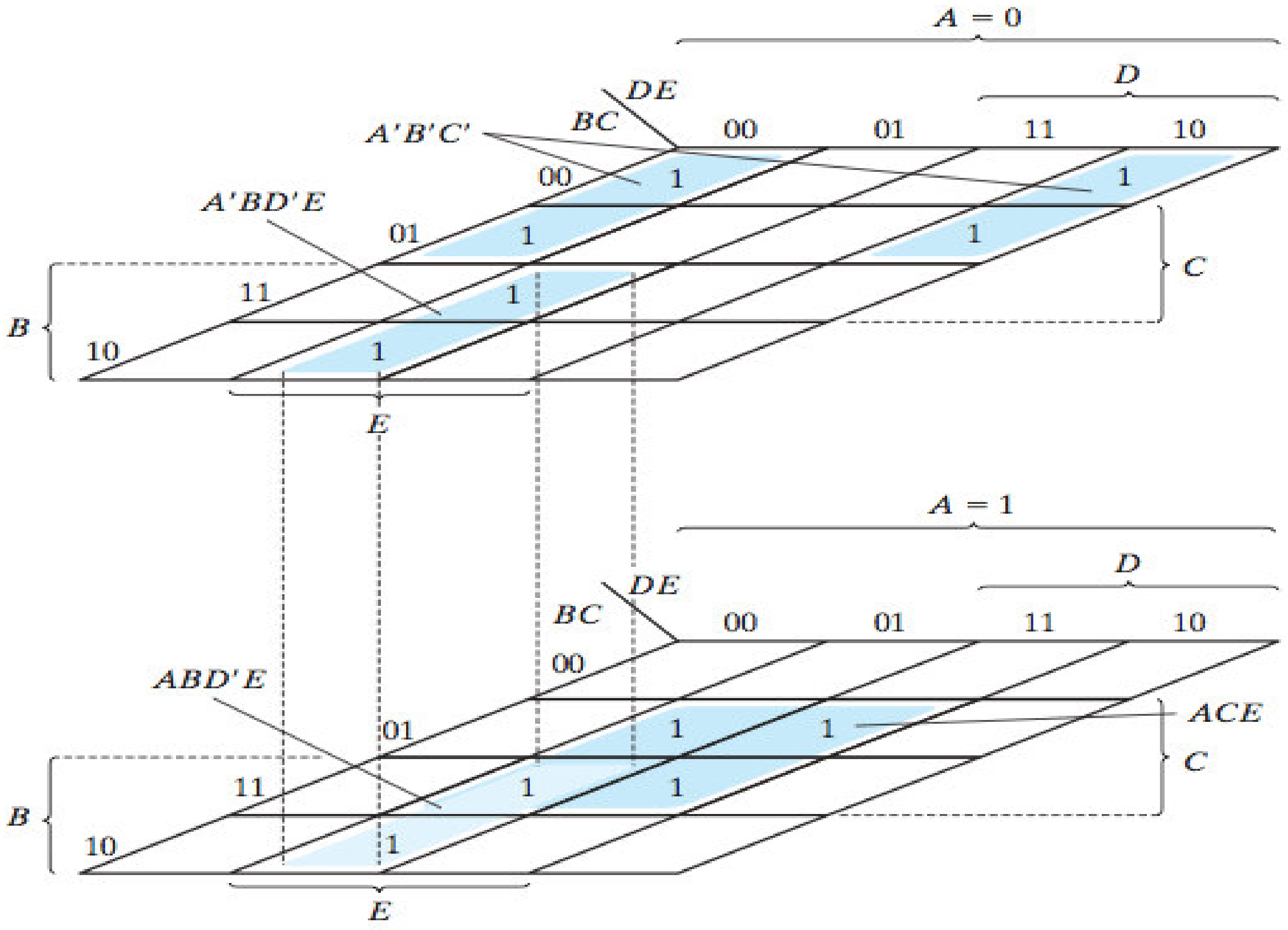


| CD \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 1 |
| 01 | 1 | 1 | 1 | 0 |
| 11 | 1 | 1 | 1 | 0 |
| 10 | 0 | 0 | 0 | 0 |

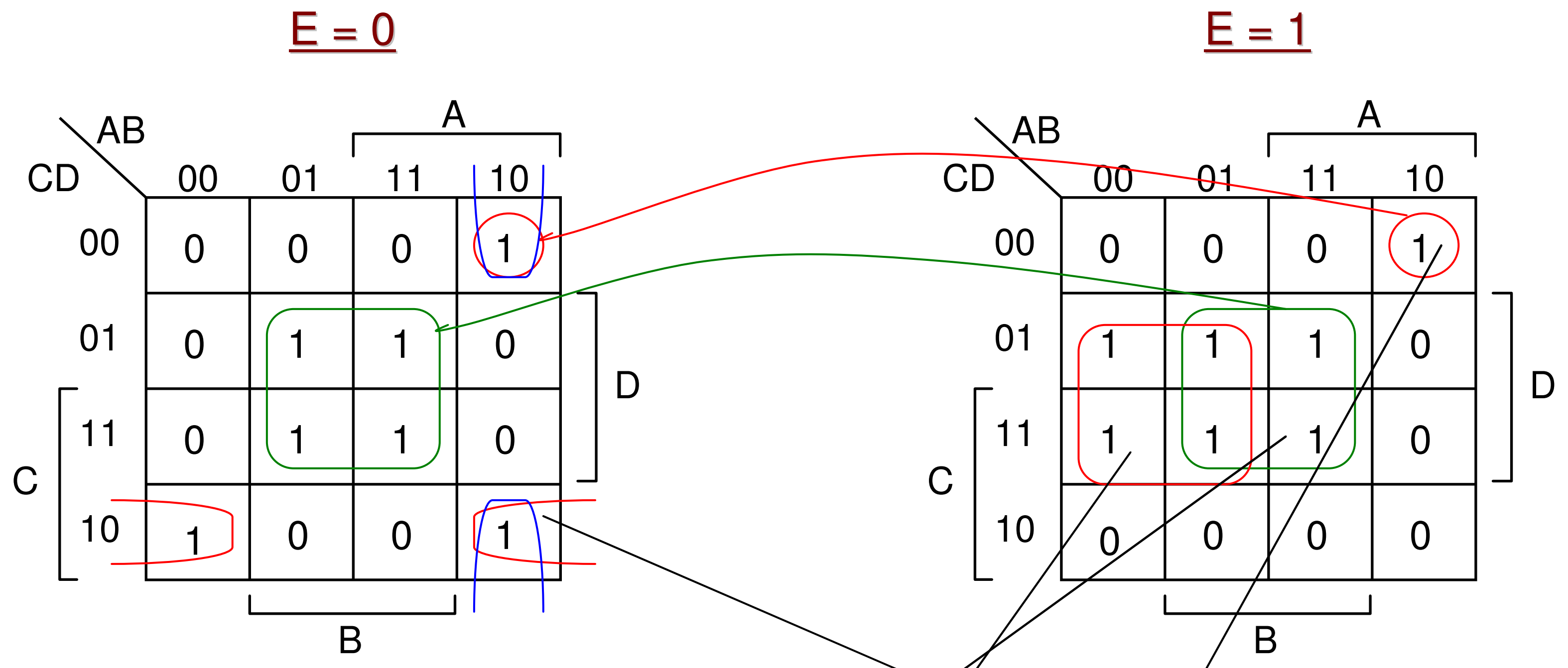
**E = 1**

$$f(A,B,C,D,E) = \square m\ (3,4,7,10,11,14,15,16,17,20,26,27,30\ 31)$$

## 5 Variable K-Maps



# 5 Variable K-Maps



**E = 0**

| CD \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 1 |
| 01 | 0 | 1 | 1 | 0 |
| 11 | 0 | 1 | 1 | 0 |
| 10 | 1 | 0 | 0 | 1 |

**E = 1**

| CD \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 1 |
| 01 | 1 | 1 | 1 | 0 |
| 11 | 1 | 1 | 1 | 0 |
| 10 | 0 | 0 | 0 | 0 |

$$F(A, B, C, D, E) = B\,D + \overline{A}\,D\,E + A\,\overline{B}\,\overline{C}\,\overline{D} + \overline{B}\,C\,\overline{D}\,\overline{E}$$

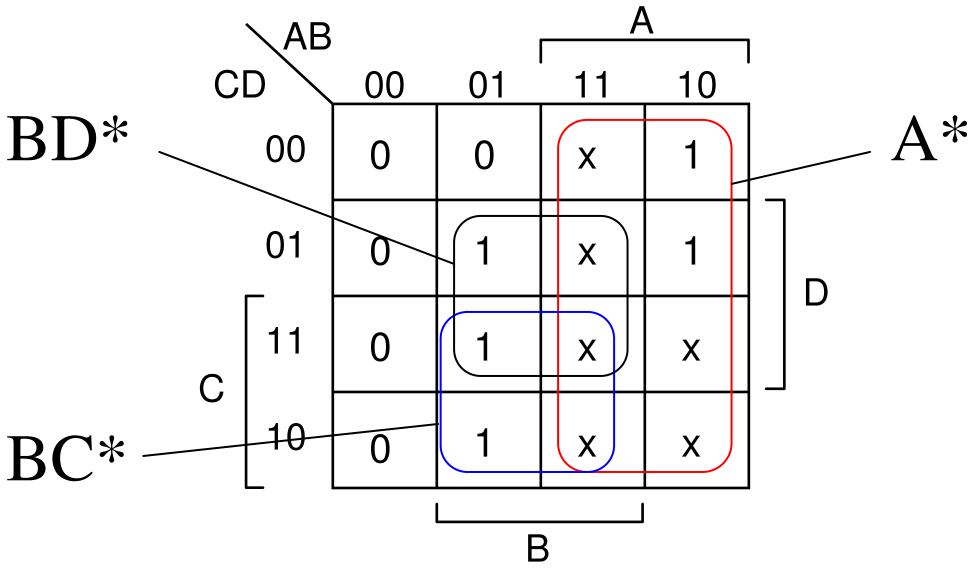
F (A,B,C,D,E) = □m (3, 4, 7, 10, 11, 14, 15, 16, 17, 20, 26, 27, 30, 31)

## Don’t Cares

- For expression minimization, don’t care values (“-” or “x”) can be assigned either 0 or 1
  - Hard to use in algebraic simplification; must evaluate all possible combinations
  - K-map minimization easily handles don’t cares

- Basic don’t care rule for K-maps is ***include*** the “dc” (“-” or “x”) in group ***if it helps*** to form a larger group; else leave it out

## K-map Minimization with Don’t Cares



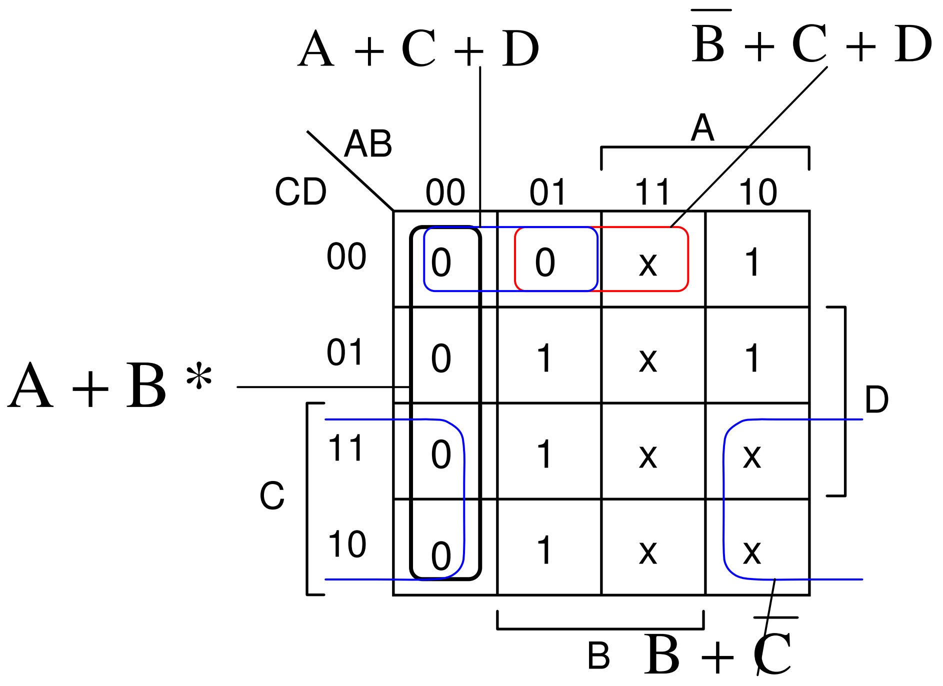
| CD \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | x | 1 |
| 01 | 0 | 1 | x | 1 |
| 11 | 0 | 1 | x | x |
| 10 | 0 | 1 | x | x |

We want a sum of products expression so we circle 1s and x’s (don’t cares)

* PIs are essential; no other implicants remain ( no secondary PIs). The minimal expression is:

$$X_3 = A + BC + BD$$

## K-map Minimization with Don't Cares



We want a product of sums expression so we circle 0s and x's (don't cares)

* PIs are essential; there are 3 secondary Pis. The minimal expressions are:

$$F = (A + B) \cdot (\overline{B} + C + D)$$

$$F = (A + B) \cdot (A + C + D)$$

## Cost Criteria

- K-Maps result with minimized sum-of-products or products-of-sums expressions
- This corresponds to 2-level circuits called AND-OR networks or OR-AND Networks.
- It is usually assumed that inputs and their complerments are available
- 3 or higher input AND, OR gates possible

  (Fan-in: number of inputs to a gate)

## Cost Criteria

- Consider $F=X.Y + X.Z + Y.Z$ (1)
- Which can be simplified to:

  $F=X.(Y + Z) + Y.Z$ (2)

Cost of these circuits may be defined as:
Literal Cost: Number of Literal appearances in the expression

(1) has a cost of : 6 literals
(2) has a cost of : 5 literals

Gate input Cost: Number of inputs to the gates

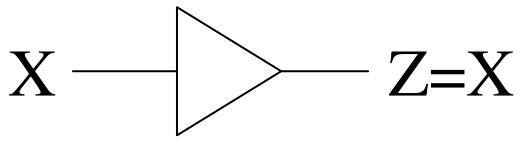
(1): 9 (2): 8

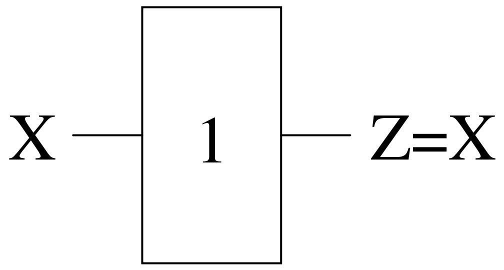
So, (2) is less costly (but slower since 3 gate delays!)

## Additional Logic Operations

- For two inputs, there are 16 ways we can assign output values
  - Besides AND and OR, five others are useful
- The unary *Buffer* operation is useful in the real world

| X | Z=X |
|---|---|
| 0 | 0 |
| 1 | 1 |

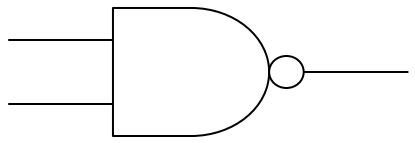


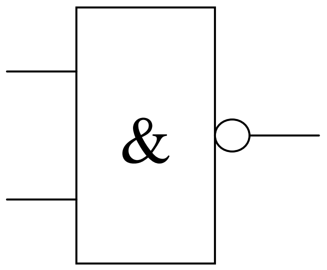


## Additional Logic Operations - NAND

- NAND (NOT - AND) is the complement of the AND operation

| X | Y | $\overline{X \cdot Y}$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

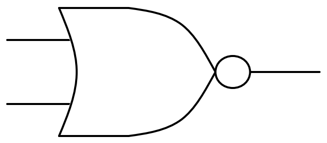


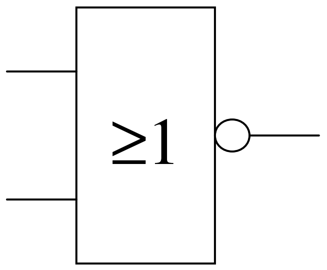


## Additional Logic Operations - NOR

- NOR (NOT - OR) is the complement of the OR operation

| X | Y | $\overline{X+Y}$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

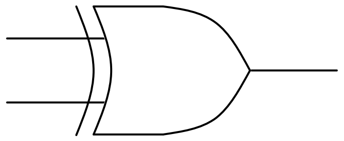


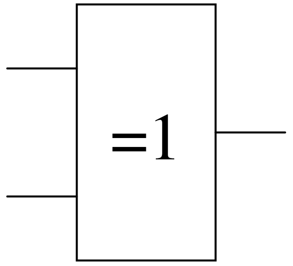


## Additional Logic Operations - XOR

- Exclusive OR is similar to the inclusive OR except output is 0 for 1, 1 inputs
- Alternatively the output is 1 when modulo 2 input sum is equal to 1

| X | Y | $X \oplus Y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

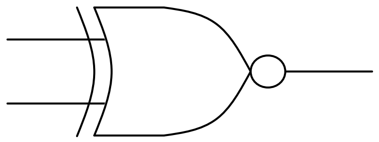


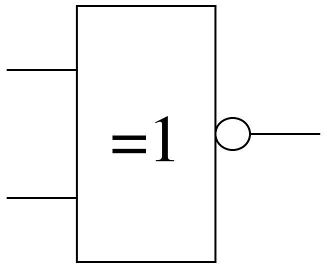


## Additional Logic Operations - XNOR

- Exclusive NOR is the complement of the XOR operation
- Alternatively the output is 1 when modulo 2 input sum is not equal to 1

| X | Y | $\overline{X \oplus Y}$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

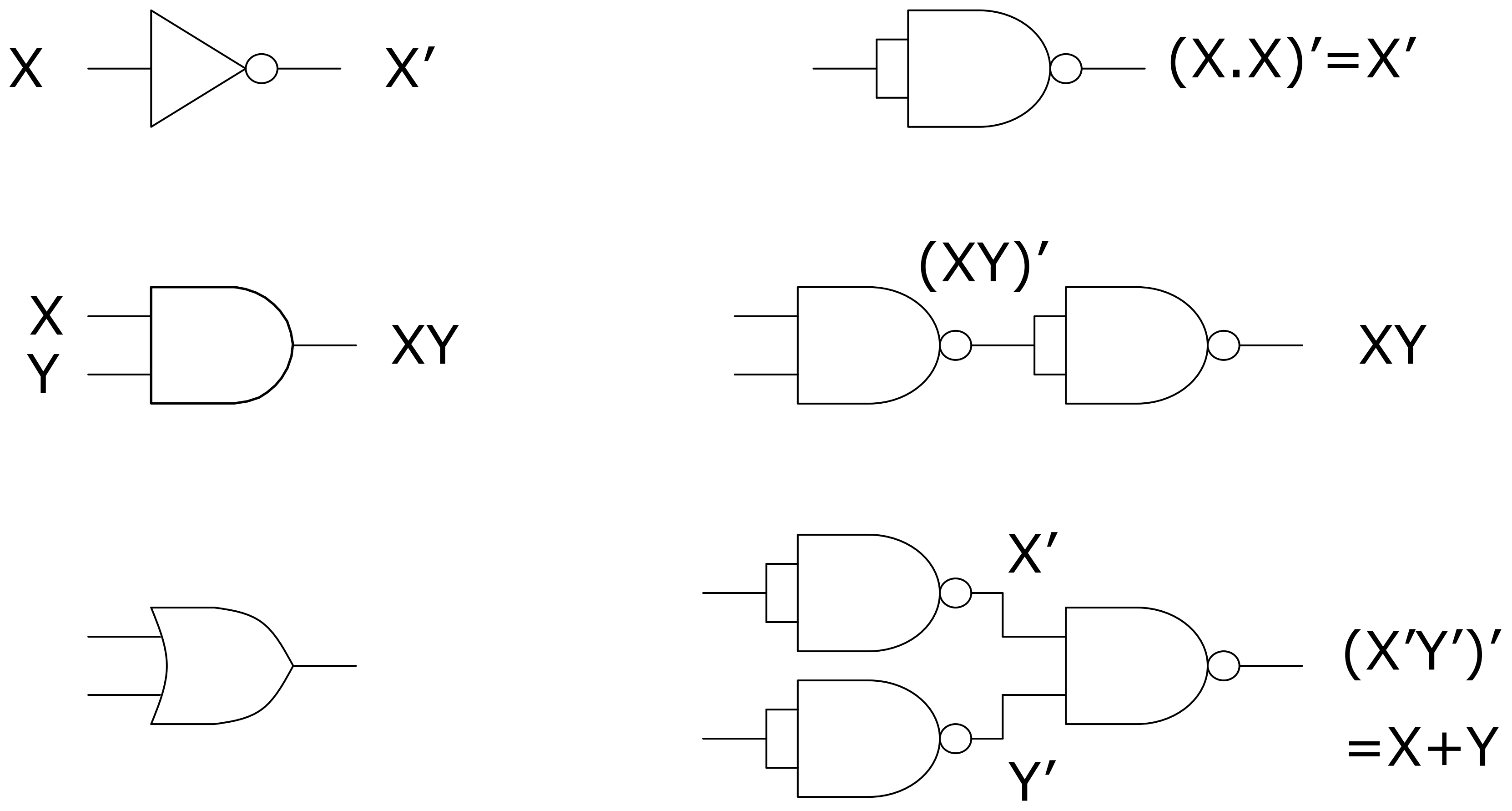




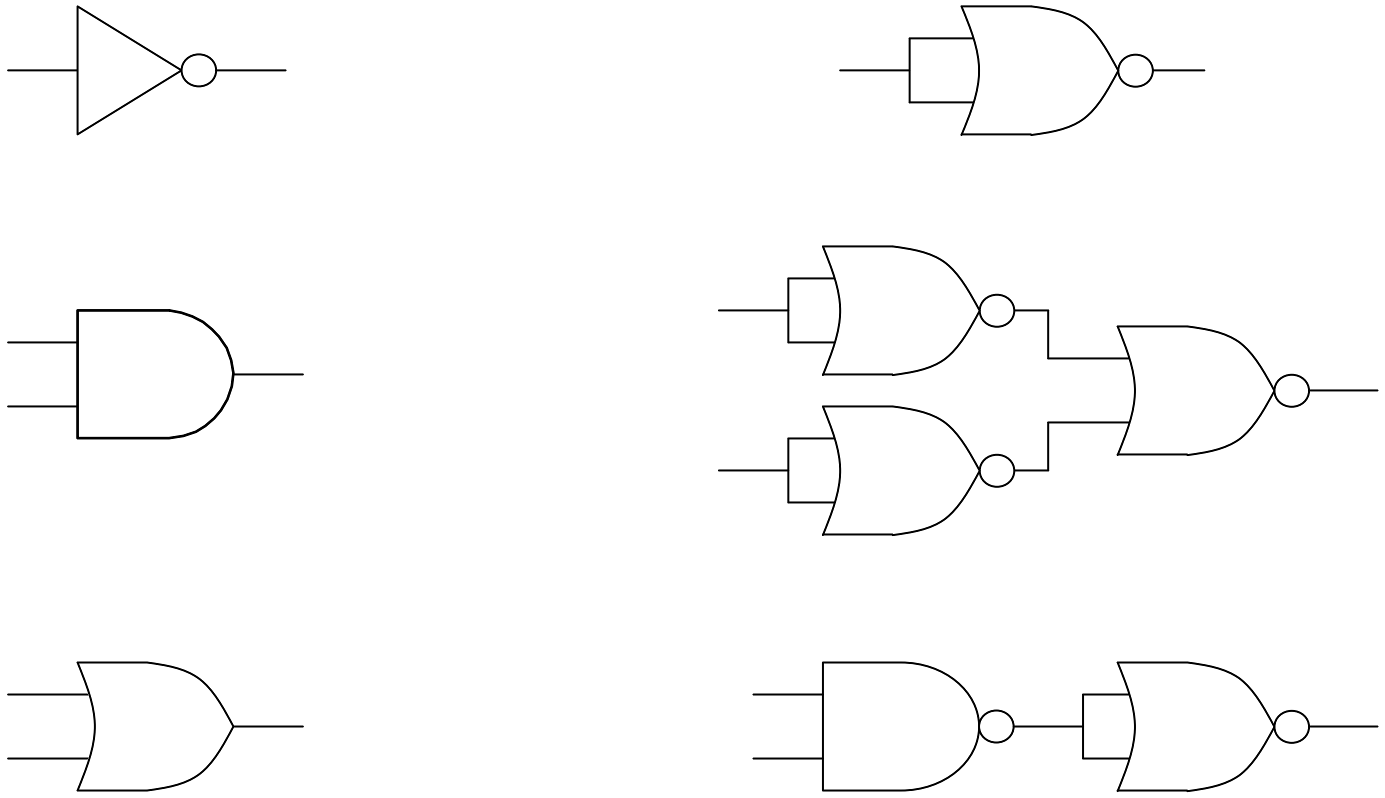
## Minimal Logic Operator Sets

- AND , OR, NOT are all that's needed to express any combinational logic function as switching algebra expression
  - operators are all that were originally defined
- Two other minimal logic operator sets exist
  - Just NAND gates
  - Just NOR gates
- We can demonstrate how just NANDs or NORs can do AND, OR, NOT operations

## NAND as a Minimal Set

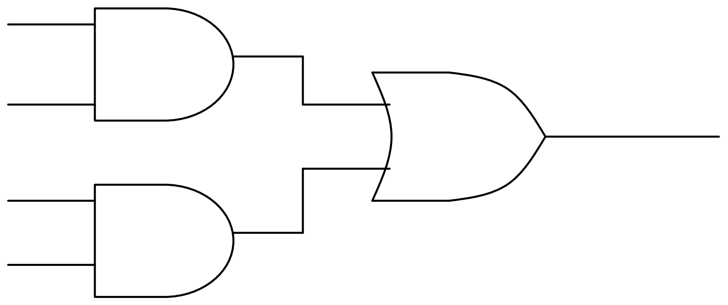


## NOR as a Minimal Set

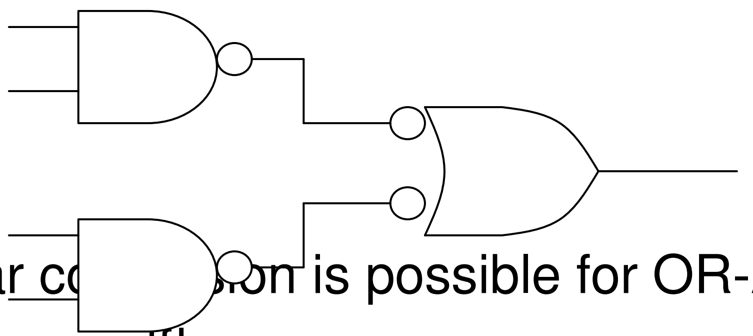


# Convert from AND-OR to NAND-NAND

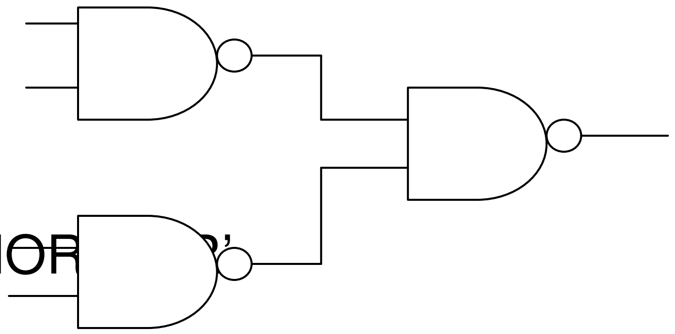
- Its easy to do the conversion:
- 1.



- 2.



3.



- Similar co[illegible]sion is possible for OR-AND to NOR[illegible]R’
- Do it yourself!

# Odd-Function

- Remember XOR function

$$X \oplus Y$$

- 1 When X,Y different
- Consider 3-input XOR:
- $(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z) = X \oplus Y \oplus Z$
- K-Map for 3-input XOR:

| Z \ XY | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | 1 | | 1 |
| 1 | 1 | | 1 | |

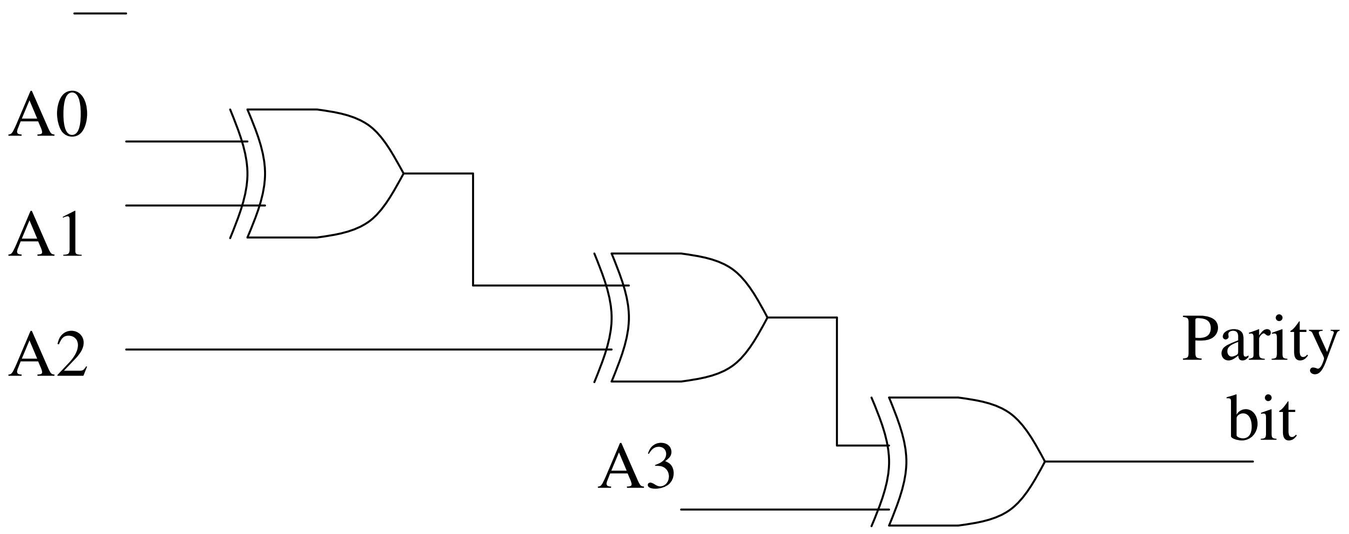
$$=(XY'+X'Y) \oplus Z$$

$$= (XY'+X'Y)\,Z' + (\overline{X}Y'+X'Y)'Z$$

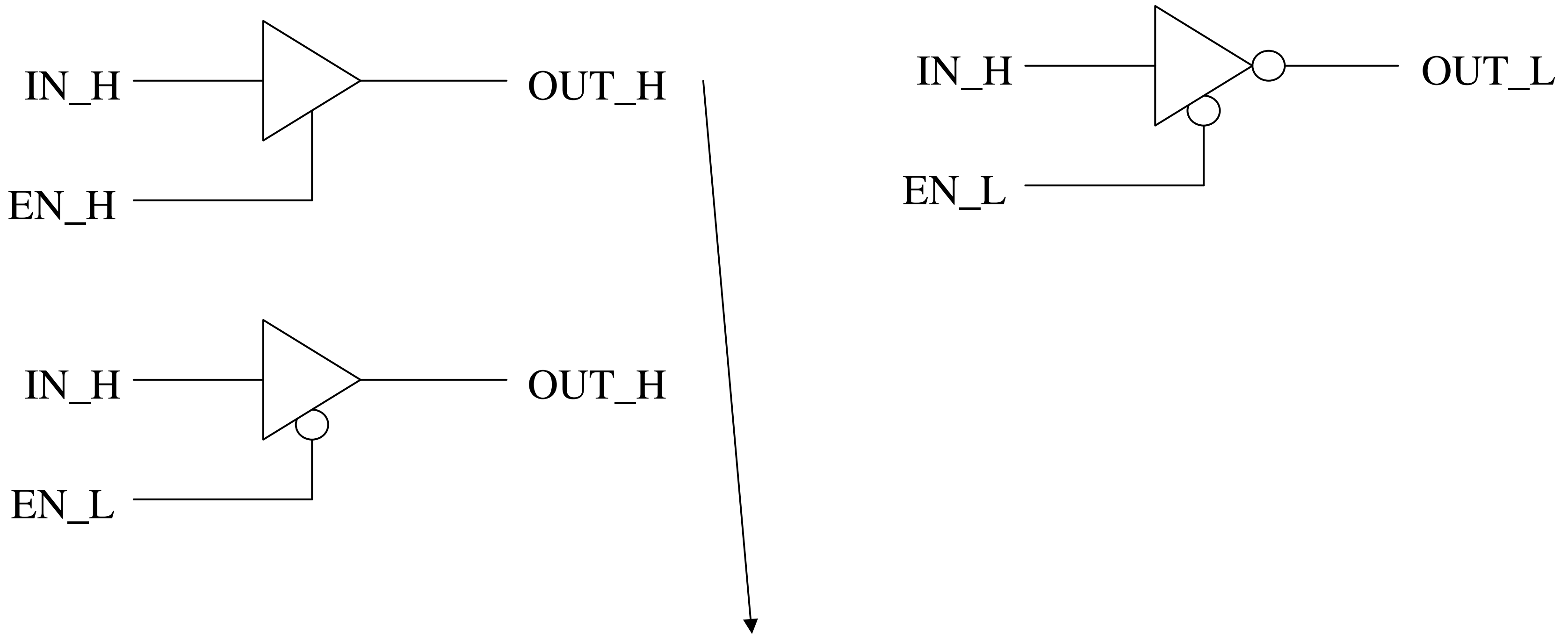
$$=XY'Z'+X'YZ'+XY'Z+X'YZ$$

## Odd-Function Cont.

- Output is ‘1’ when an odd number of inputs are ‘1’. Why is it important?
- Parity bit for even parity in ASCII code will use this function..
- For n bits, same structure continues..



# Three State Buffers



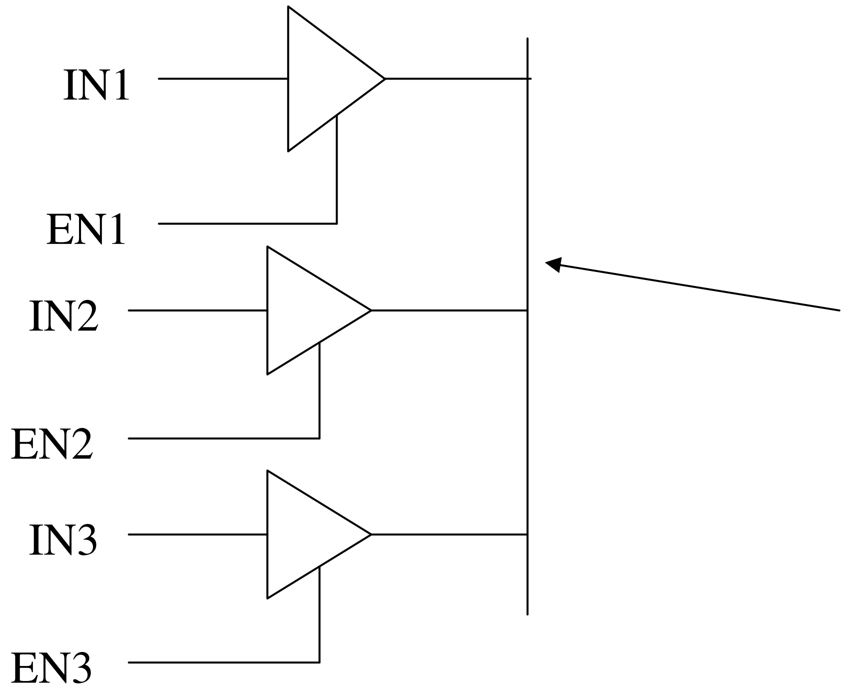
| IN | EN | OUT |
|---|---|---|
| X | 0 | HI-Z |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

High-impedance state(like an open circuit)

Same as input

## Three-State Buffers Cont.

- Three-state buffers are used usually when you want to connect only one of k inputs to a wire



Only one of IN1,IN2, IN3 will be connected , so only one of EN1, EN2, EN3 should be 1 at the same time!

Otherwise, the circuit will be burned.

## Three State Outputs

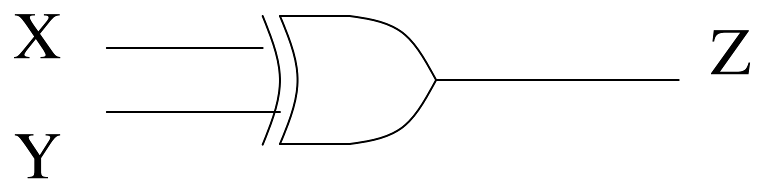
- Standard logic gate outputs only have two states; high and low
  - Outputs are effectively either connected to +V or ground (low impedance)
- Certain applications require a logic output that we can "turn off" or disable
  - Output is disconnected (high impedance)
- This is the three-state output
  - May be stand-alone (a buffer) or part of another function output

## Documenting Combinational Systems

- Schematic (circuit) diagrams are a graphical representation of the combinational circuit
  - Best practice is to organize drawing so data flows left to right, control, top to bottom
- Two conventions exist to denote circuit signal connections
  - Only ‘T’ intersections are connections; others just cross over
  - Solid dots ‘•’ are placed at connection points (This is preferred)

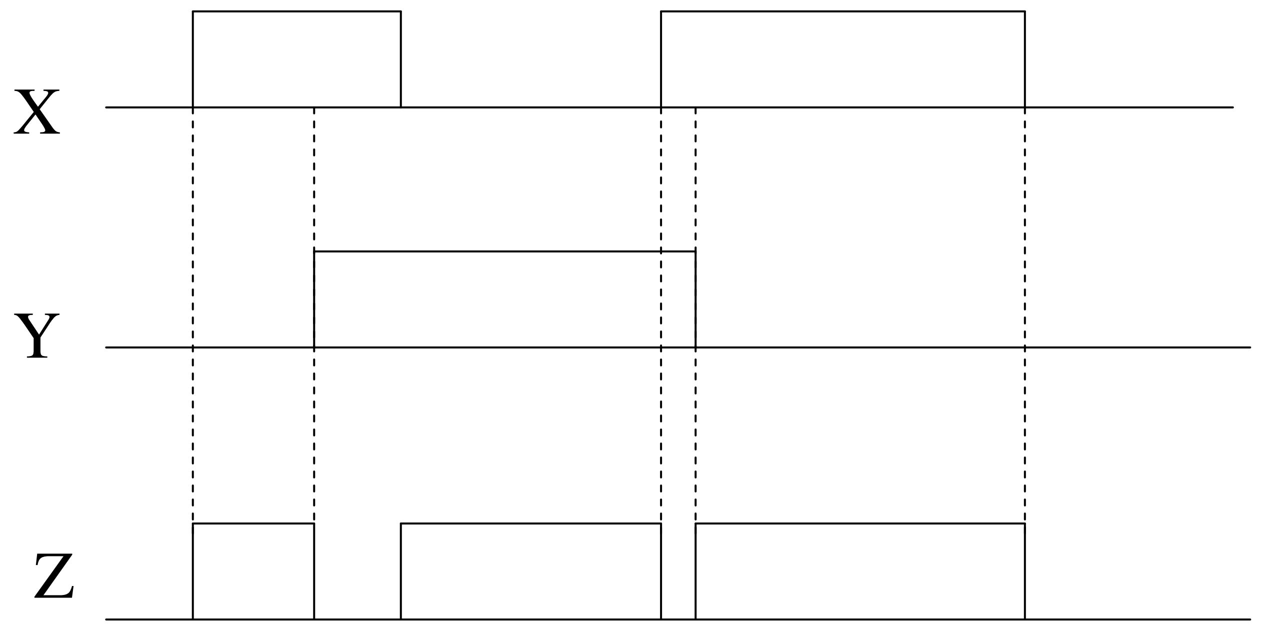
## Timing Diagrams

- Timing diagrams show how inputs and outputs for a logic circuit change in time
- For given input timing diagrams, we can draw the output timing diagrams for a given combinational circuit.
- Example:Consider the following XOR circuit with 2 inputs:



## Timing Diagrams Continued

- Timing Diagram for X (given),Y (given), and Z



- Actually, there is a delay at each new rising/falling pulse, but not shown here! (will be discussed later)