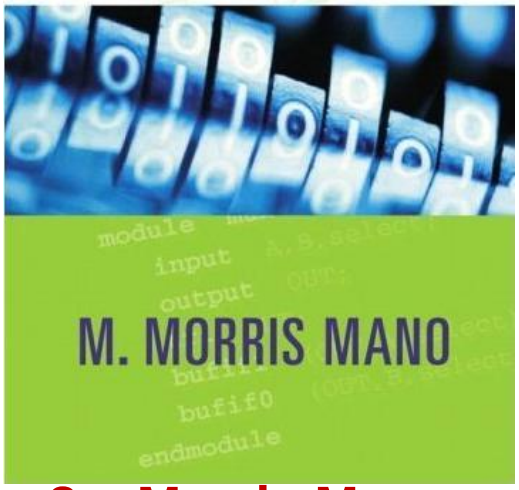


Digital Design 3e, Morris Mano

Chapter 3 – Gate Level Minimization

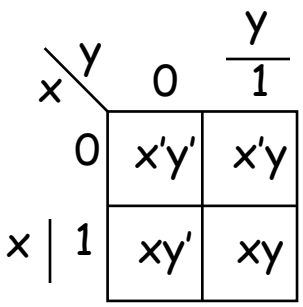
COMBINATIONAL CIRCUITS

Overview

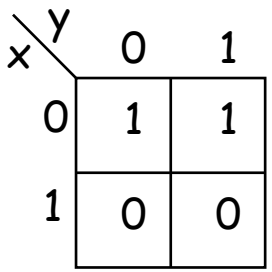
- K-maps: an alternate approach to representing Boolean functions
- K-map representation can be used to minimize Boolean functions
- Easy conversion from truth table to K-map to minimized SOP representation.
- Simple rules (steps) used to perform minimization
- Leads to minimized SOP representation.
  - Much faster and more more efficient than previous minimization techniques with Boolean algebra.

Karnaugh maps

- **Alternate way of representing Boolean function**
  - **All rows of truth table represented with a square**
  - **Each square represents a minterm**
- **Easy to convert between truth table, K-map, and SOP**
  - **Unoptimized form: number of 1's in K-map equals number of minterms (products) in SOP**
  - **Optimized form: reduced number of minterms**



$F = \Sigma(m_0, m_1) = x'y + x'y'$



x	y	F
0	0	1
0	1	1
1	0	0
1	1	0

Karnaugh Maps

- **A Karnaugh map is a graphical tool for assisting in the general simplification procedure.**
- **Two variable maps.**

A \ B	0	1
0	0	1
1	1	0

$F = AB' + A'B$

A \ B	0	1
0	0	1
1	1	1

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

- **Three variable maps.**

A \ BC	00	01	11	10
0	0	1	0	1
1	1	1	1	1

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

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

Rules for K-Maps

- We can reduce functions by circling 1's in the K-map
- Each circle represents minterm reduction
- Following circling, we can deduce minimized and-or form.

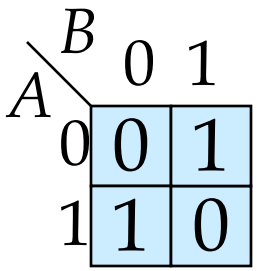
Rules to consider

- Every cell containing a 1 must be included at least once.
- The largest possible "power of 2 rectangle" must be enclosed.
- The 1's must be enclosed in the smallest possible number of rectangles.

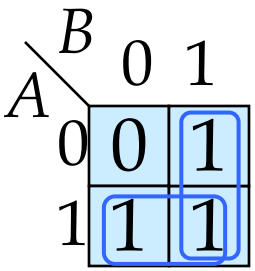
Example

Karnaugh Maps

- **A Karnaugh map is a graphical tool for assisting in the general simplification procedure.**
- **Two variable maps.**



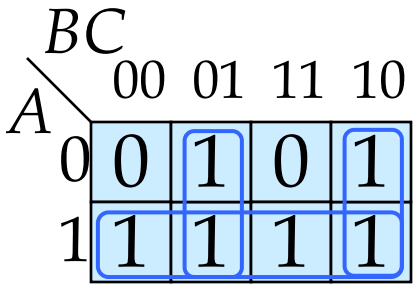
$F = AB' + A'B$



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

$F = A + B$

- **Three variable maps.**

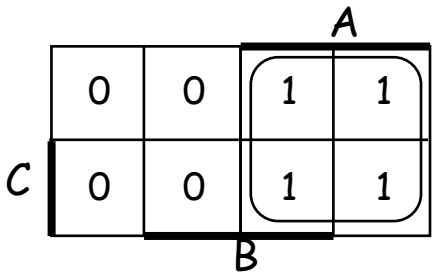


$F = A + B'C + BC'$

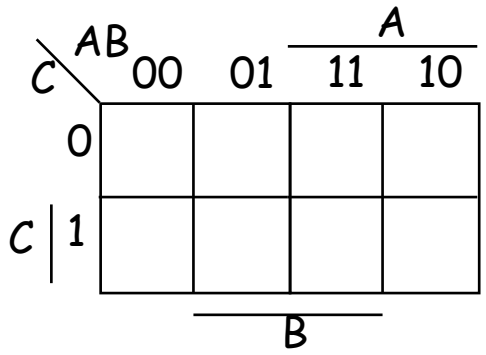
$F = AB'C' + AB'C + ABC + ABC' + A'B'C + A'BC'$

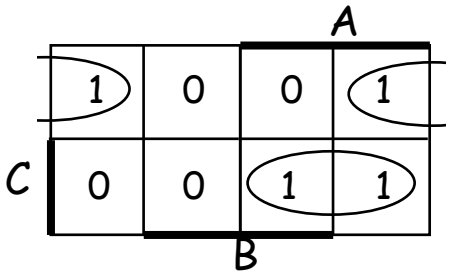
Karnaugh maps

- **Numbering scheme based on Gray–code**
  - **e.g., 00, 01, 11, 10**
  - **Only a single bit changes in code for adjacent map cells**
  - **This is necessary to observe the variable transitions**



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

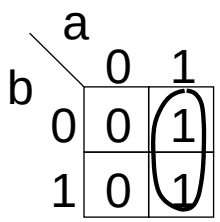




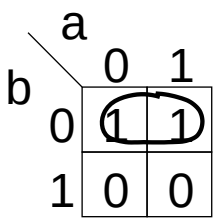
$F(A,B,C) = \Sigma m(0,4,5,7) = AC + B'C'$

More Karnaugh Map Examples

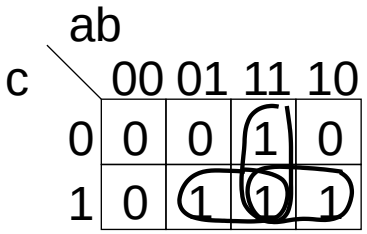
- **Examples**



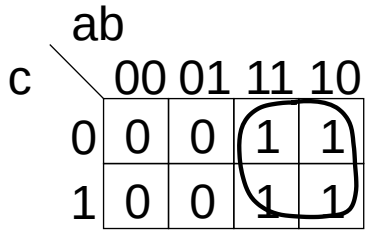
f = a



g = b'



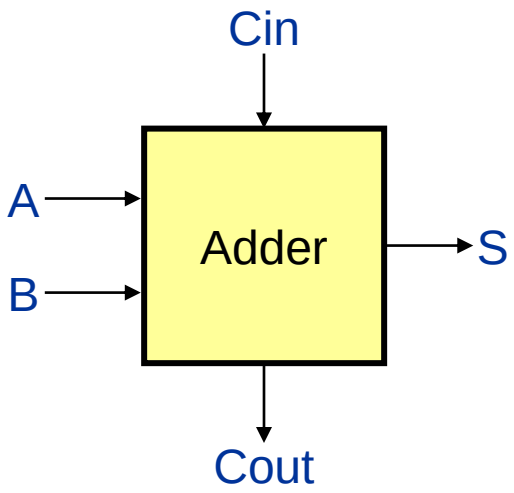
cout = ab + bc + ac



f = a

1. Circle the largest groups possible.
2. Group dimensions must be a power of 2.
3. Remember what circling means!

Application of Karnaugh Maps: The One-bit Adder



A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

How to use a Karnaugh Map instead of the Algebraic simplification?

S = A'B'Cin + A'BCin' + A'BCin + ABCin

Cout = A'BCin + A B'Cin + ABCin' + ABCin

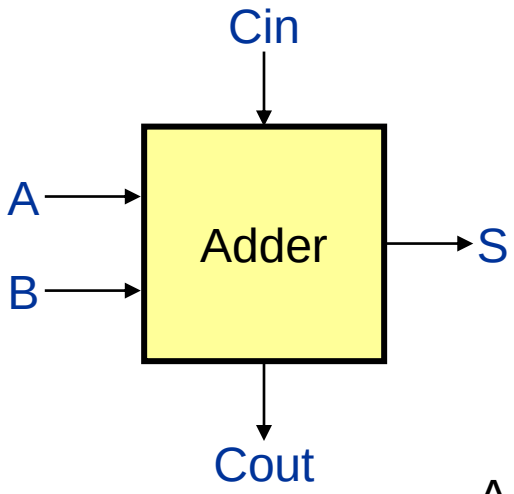
= A'BCin + ABCin + AB'Cin + ABCin + ABCin' + ABCin

= (A' + A)BCin + (B' + B)ACin + (Cin' + Cin)AB

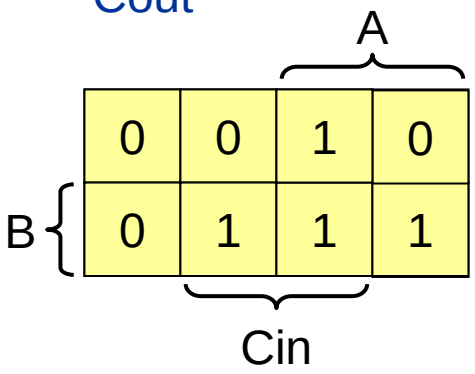
= 1·BCin + 1· ACin + 1· AB

= BCin + ACin + AB

Application of Karnaugh Maps: The One-bit Adder



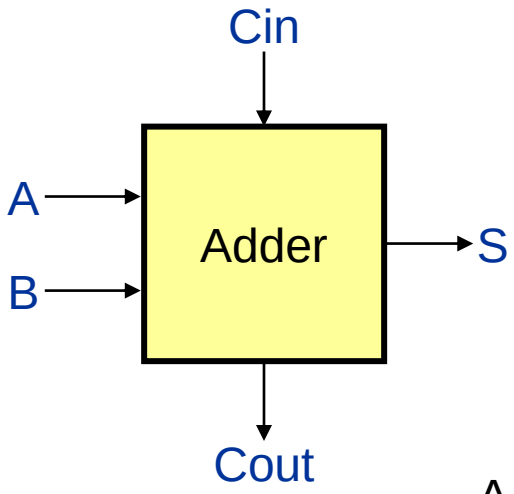
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



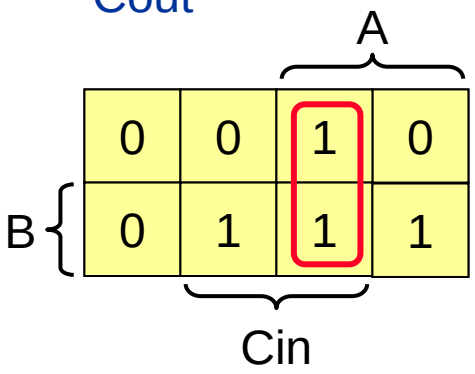
Karnaugh Map for Cout

Now we have to cover all the 1s in the Karnaugh Map using the largest rectangles and as few rectangles as we can.

Application of Karnaugh Maps: The One-bit Adder



A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

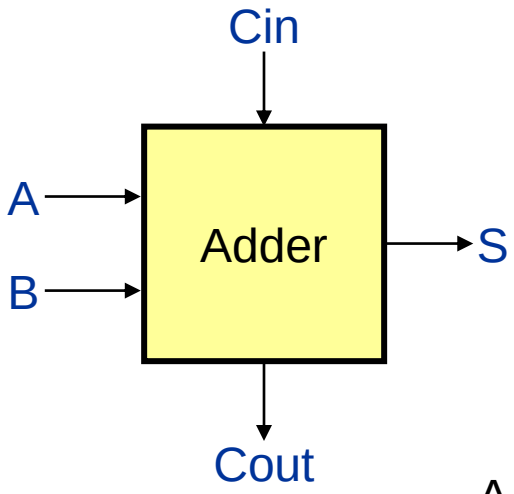


Karnaugh Map for Cout

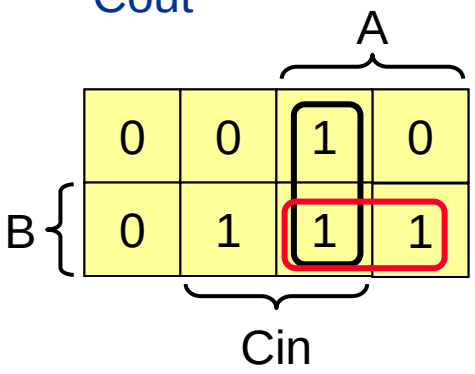
Now we have to cover all the 1s in the Karnaugh Map using the largest rectangles and as few rectangles as we can.

Cout = ACin

Application of Karnaugh Maps: The One-bit Adder



A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

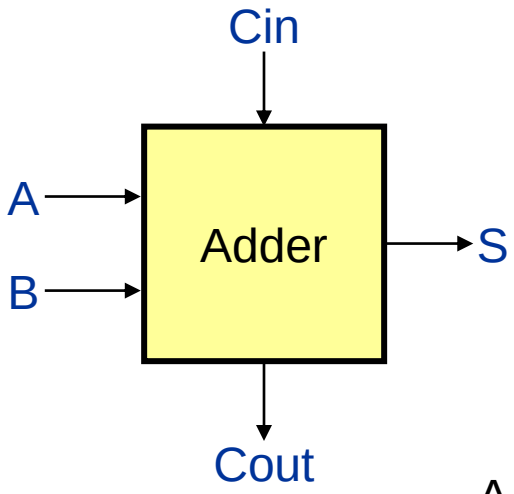


Karnaugh Map for Cout

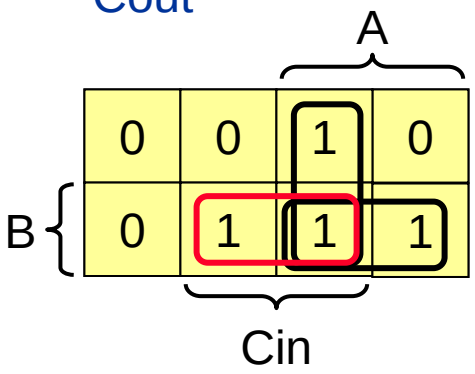
Now we have to cover all the 1s in the Karnaugh Map using the largest rectangles and as few rectangles as we can.

Cout = Acin + AB

Application of Karnaugh Maps: The One-bit Adder



A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

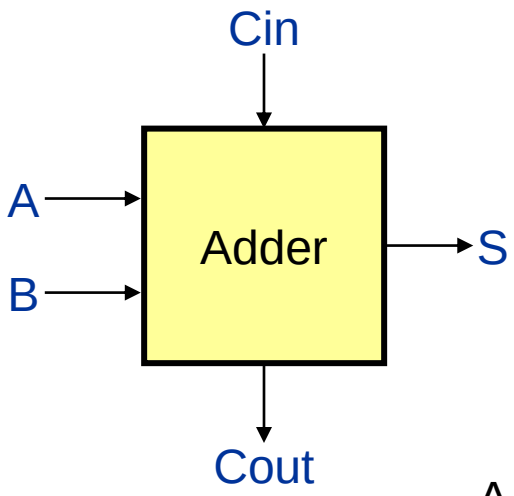


Karnaugh Map for Cout

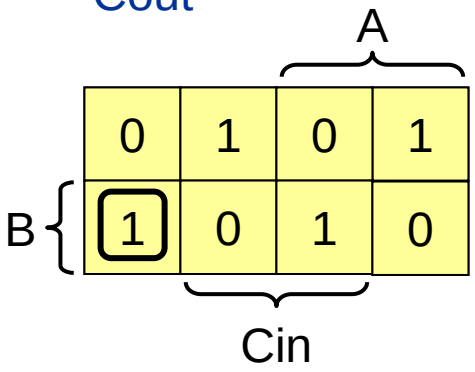
Now we have to cover all the 1s in the Karnaugh Map using the largest rectangles and as few rectangles as we can.

Cout = ACin + AB + BCin

Application of Karnaugh Maps: The One-bit Adder



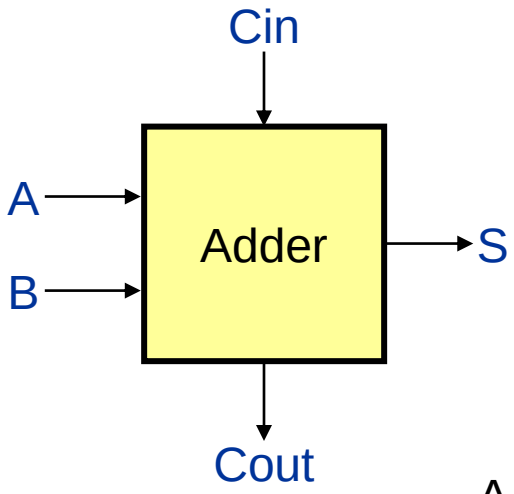
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



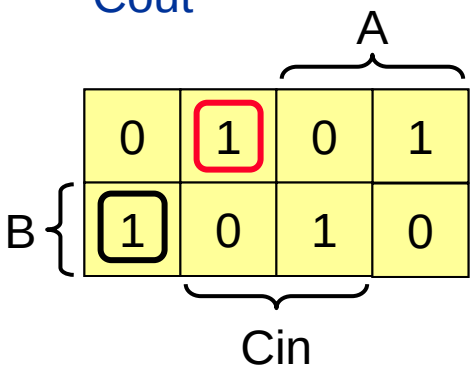
Karnaugh Map for S

S = A'BCin'

Application of Karnaugh Maps: The One-bit Adder



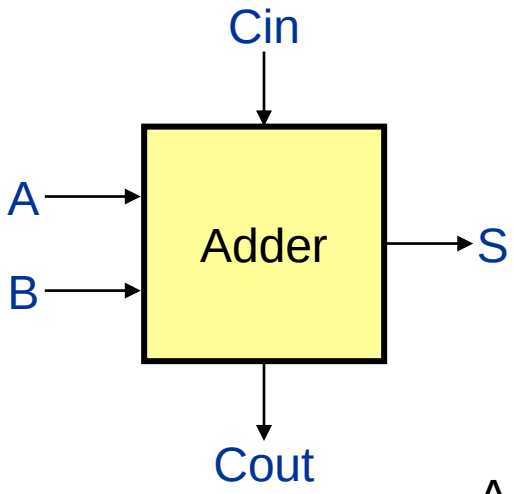
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



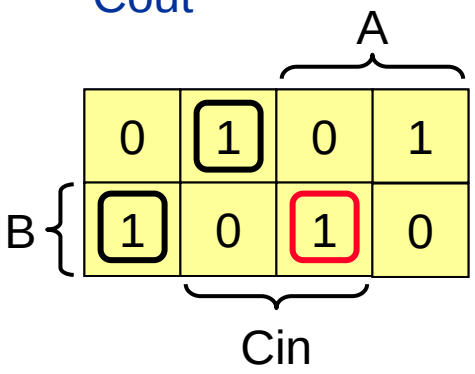
Karnaugh Map for S

S = A'BCin' + A'B'Cin

Application of Karnaugh Maps: The One-bit Adder



A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

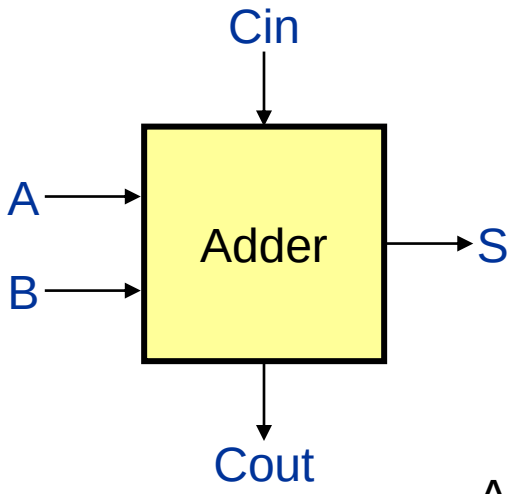


Karnaugh Map for S

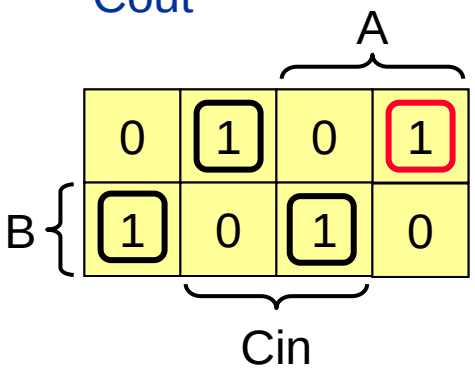
S = A'BCin' + A'B'Cin + ABCin

Application of Karnaugh Maps: The One-bit Adder

Can you draw the circuit diagrams?



A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Karnaugh Map for S

S = A'BCin' + A'B'Cin + ABCin + AB'Cin'

No Possible Reduction!

Summary

- Karnaugh map allows us to represent functions with new notation
- Representation allows for logic reduction.
  - Implement same function with less logic
- Each square represents one minterm
- Each circle leads to one product term
- Not all functions can be reduced
- Each circle represents an application of:
  - Distributive rule -- $x(y + z) = xy + xz$
  - Complement rule – $x + x' = 1$

More Karnaugh Maps and Don’t Cares

Overview

- Karnaugh maps with four inputs
  - Same basic rules as three input K-maps
- Understanding prime implicants
  - Related to minterms
- Covering all implicants
- Using Don’t Cares to simplify functions
  - Don’t care outputs are undefined
- Summarizing Karnaugh maps

Karnaugh Maps for Four Input Functions

- **Represent functions of 4 inputs with 16 minterms**
- **Use same rules developed for 3-input functions**
- **Note bracketed sections shown in example.**

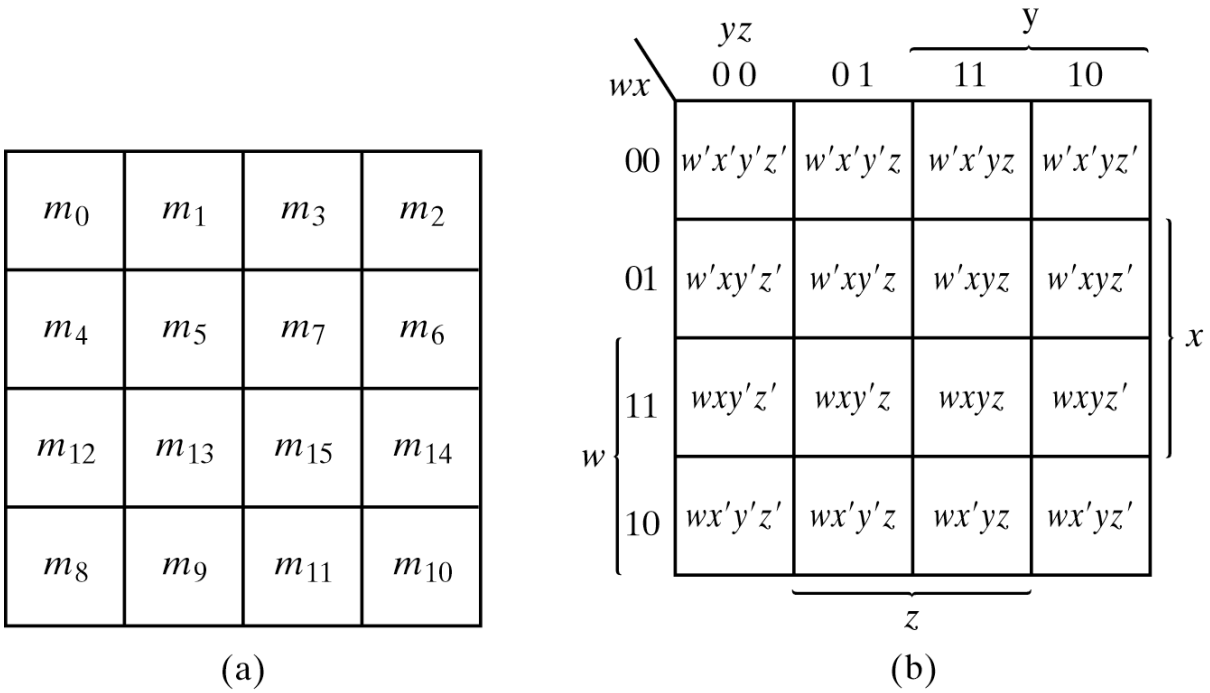


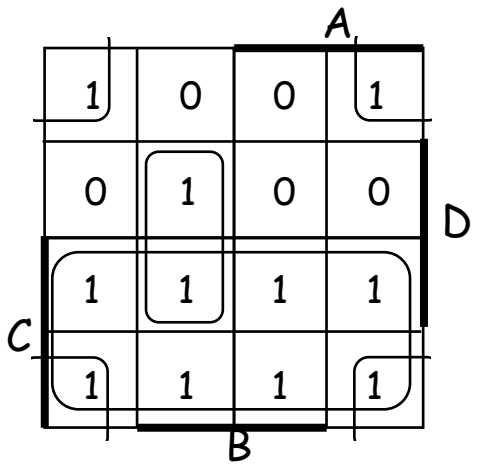
Fig. 3-8 Four-variable Map

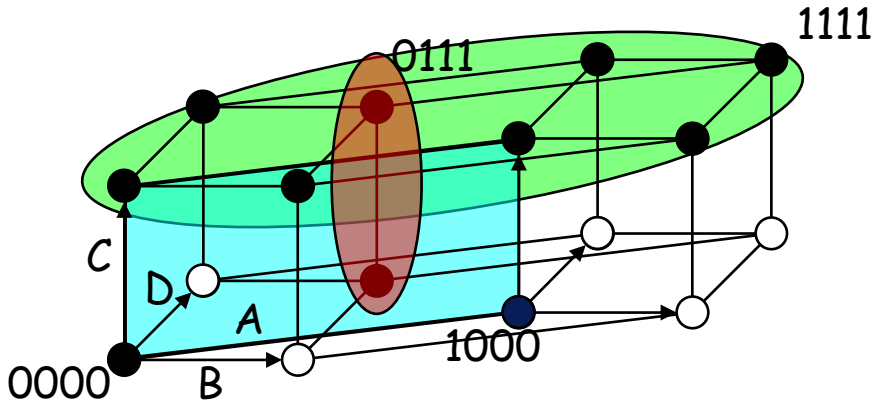
Karnaugh map: 4-variable example

° **F(A,B,C,D) = Σm(0,2,3,5,6,7,8,10,11,14,15)**

F =

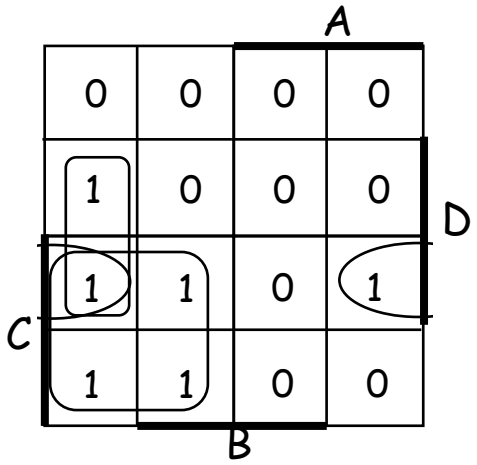
C + A'BD + B'D'



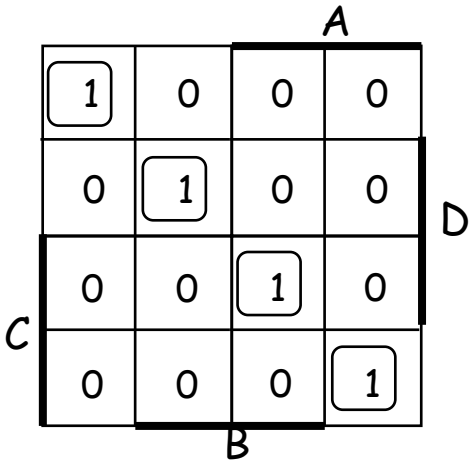


Solution set can be considered as a coordinate System!

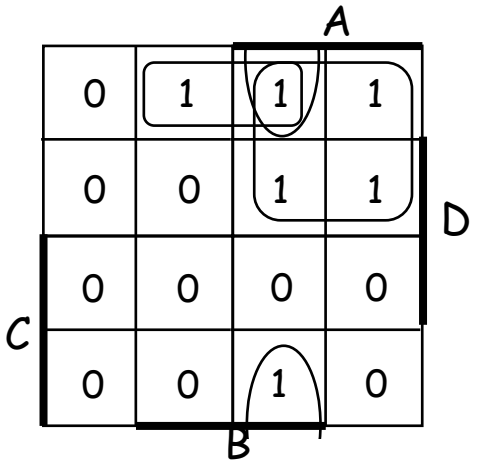
Design examples



K-map for LT



K-map for EQ



K-map for GT

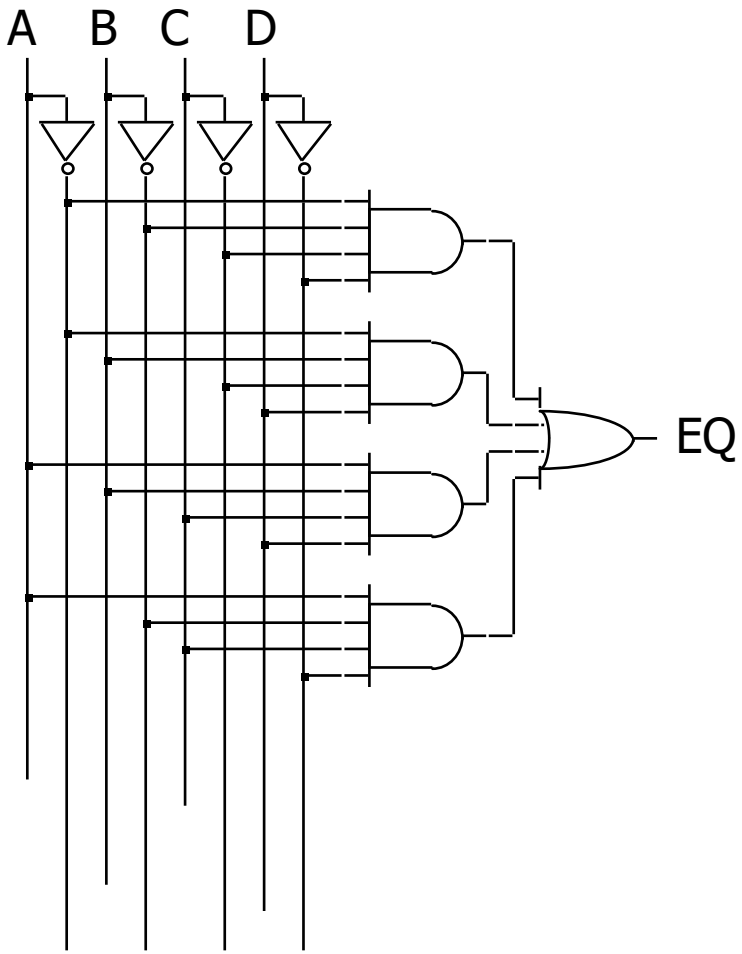
LT = A' B' D + A' C + B' C D

EQ = A'B'C'D' + A'BC'D + ABCD + AB'CD'

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

Can you draw the truth table for these examples?

Physical Implementation



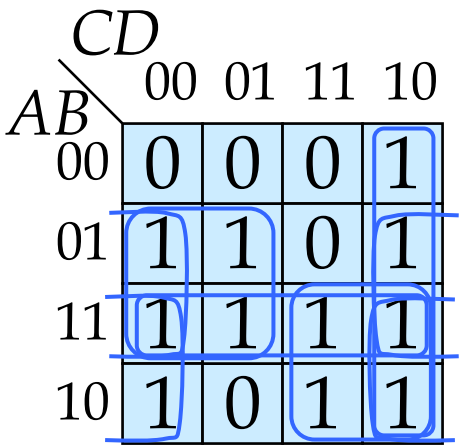
- Step 1: Truth table
- Step 2: K-map
- Step 3: Minimized sum-of-products
- Step 4: Physical implementation with gates

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

K-map for EQ

Karnaugh Maps

- **Four variable maps.**



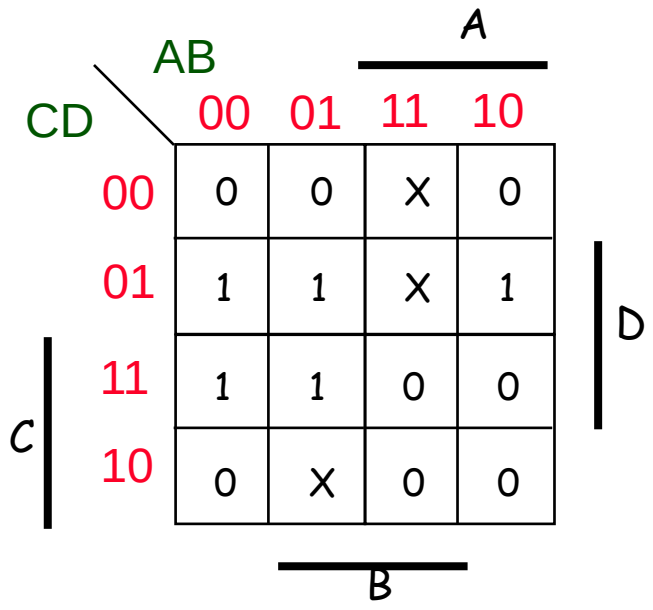
$$F=A'BC'+A'CD'+ABC$$
$$+AB'C'D'+ABC'+AB'C$$

$$F=BC'+CD'+AC+AD'$$

- **Need to make sure all 1's are covered**
- **Try to minimize total product terms.**
- **Design could be implemented using NANDs and NORs**

Karnaugh maps: Don't cares

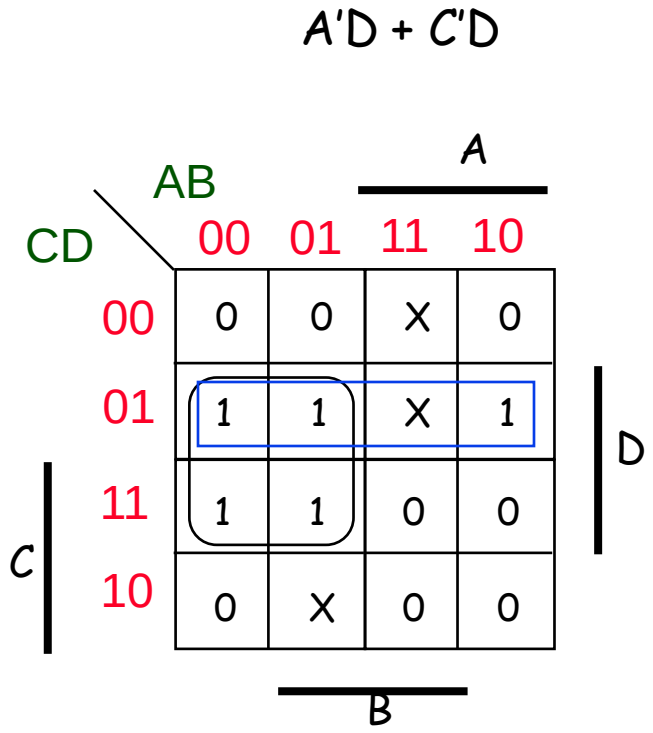
- In some cases, outputs are undefined
- We "don't care" if the logic produces a 0 or a 1
- This knowledge can be used to simplify functions.



- Treat X's like either 1's or 0's
- Very useful
- OK to leave some X's uncovered

Karnaugh maps: Don’t cares

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

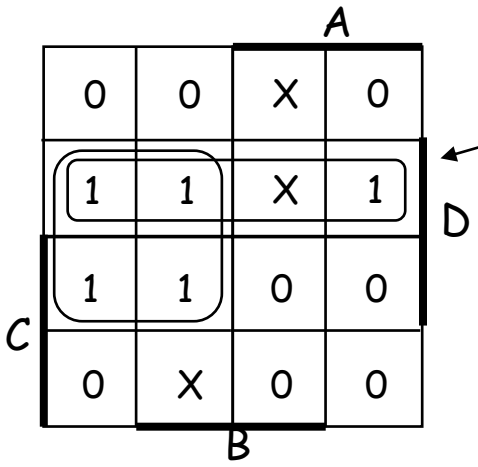


A	B	C	D	f
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	X
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	X
1	1	0	1	X
1	1	1	0	0
1	1	1	1	0

Karnaugh maps: don’t cares (cont’d)

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

A'D + C'D



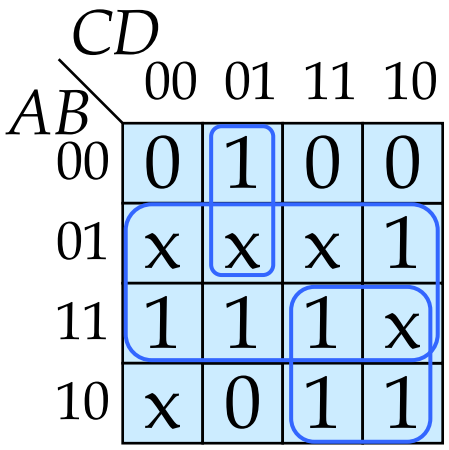
by using don't care as a "1"
a 2-cube can be formed
rather than a 1-cube to cover
this node

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

Don’t Care Conditions

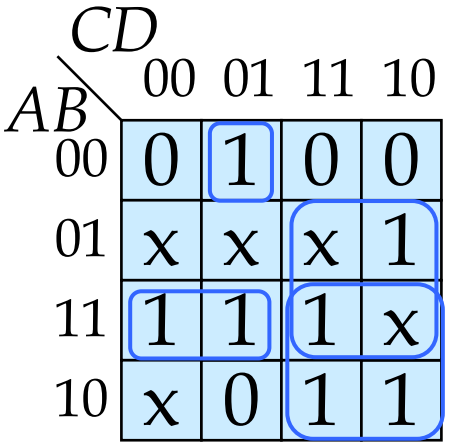
- **In some situations, we don’t care about the value of a function for certain combinations of the variables.**
  - **these combinations may be impossible in certain contexts**
  - **or the value of the function may not matter in when the combinations occur**
- **In such situations we say the function is *incompletely specified* and there are multiple (completely specified) logic functions that can be used in the design.**
  - **so we can select a function that gives the simplest circuit**
- **When constructing the terms in the simplification procedure, we can choose to either cover or not cover the don’t care conditions.**

Map Simplification with Don't Cares



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

° **Alternative covering.**

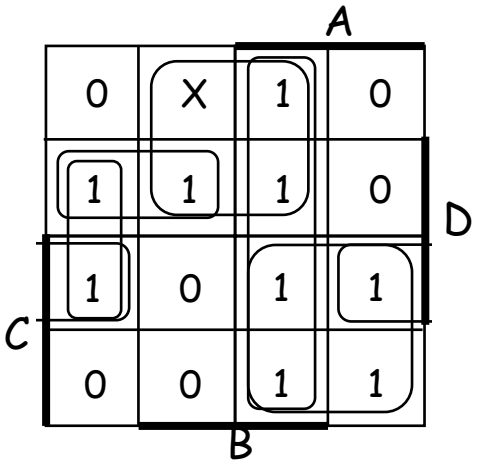


$F=A'B'C'D+ABC'+BC+AC$

Definition of terms for two-level simplification

- **Implicant**
  - Single product term of the ON-set (terms that create a logic 1)
- **Prime implicant**
  - Implicant that can't be combined with another to form an implicant with fewer literals.
- **Essential prime implicant**
  - Prime implicant is essential if it alone covers a minterm in the K-map
  - Remember that all squares marked with 1 must be covered
- **Objective:**
  - Grow implicant into prime implicants (minimize literals per term)
  - Cover the K-map with as few prime implicants as possible (minimize number of product terms)

Examples to illustrate terms



6 prime implicants:

A'B'D, BC', AC, A'C'D, AB, B'CD

essential

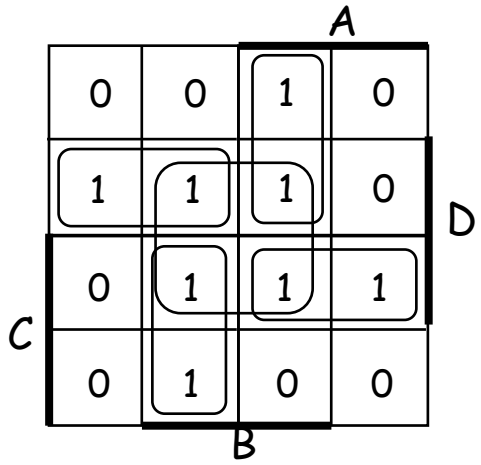
minimum cover: AC + BC' + A'B'D

5 prime implicants:

BD, ABC', ACD, A'BC, A'C'D

essential

minimum cover: 4 essential implicants

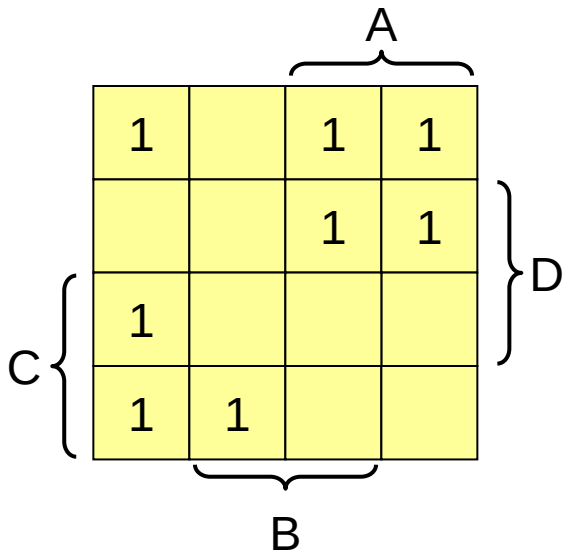


Prime Implicants

Any single 1 or group of 1s in the Karnaugh map of a function F is an implicant of F.

A product term is called a prime implicant of F if it cannot be combined with another term to eliminate a variable.

Example:



If a function F is represented by this Karnaugh Map. Which of the following terms are implicants of F, and which ones are prime implicants of F?

(a) AC'D'
(b) BD
(c) A'B'C'D'
(d) AC'
(e) B'C'D'

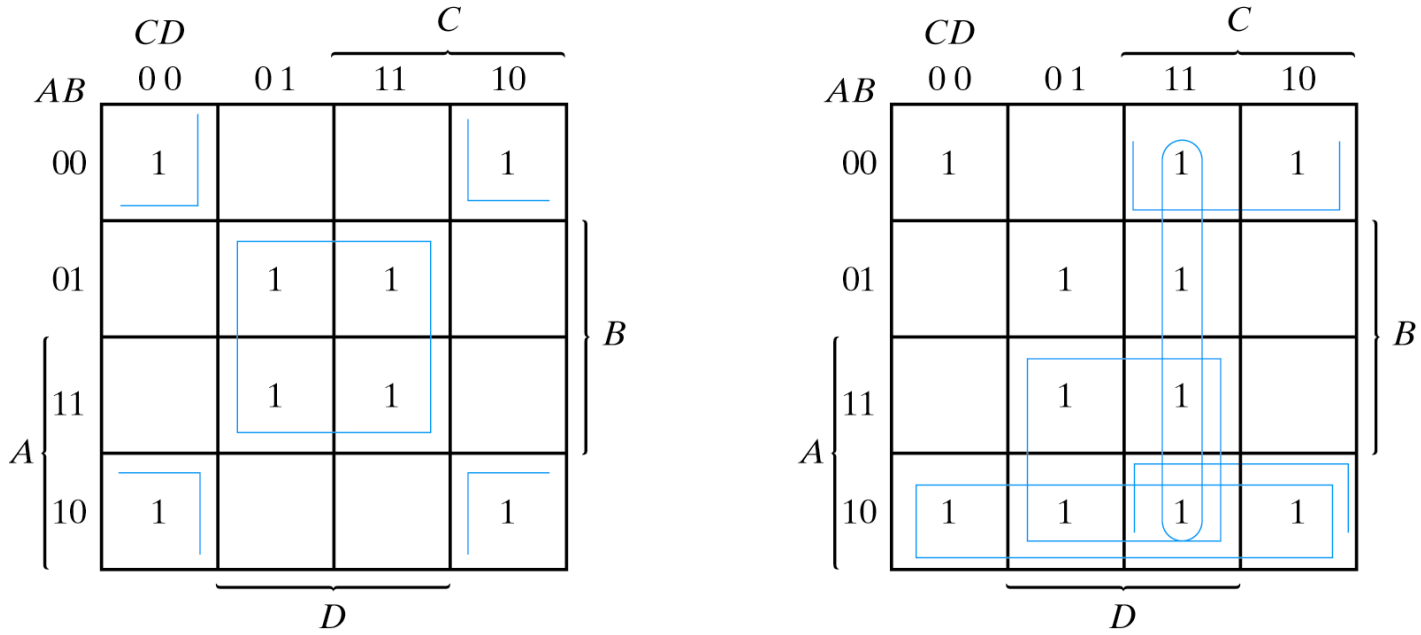
Implicants: (a),(c),(d),(e)

Prime Implicants: (d),(e)

Essential Prime Implicants

A product term is an essential prime implicant if there is a minterm that is only covered by that prime implicant.

- The minimal sum-of-products form of F must include all the essential prime implicants of F.



(a) Essential prime implicants BD and B′D′

(b) Prime implicants CD, B′C AD, and AB′

Fig. 3-11 Simplification Using Prime Implicants

Solution

$$
\begin{aligned}
F &= BD + B'D' + CD + AD \\
&= BD + B'D' + CD + AB' \\
&= BD + B'D' + B'C + AD \\
&= BD + B'D' + B'C + AB'
\end{aligned}
$$

Summary

- K-maps of four literals considered
  - Larger examples exist
- Don’t care conditions help minimize functions
  - Output for don’t cares are undefined
- Result of minimization is minimal sum-of-products
- Result contains prime implicants
- Essential prime implicants are required in the implementation

NAND and XOR Implementations

Overview

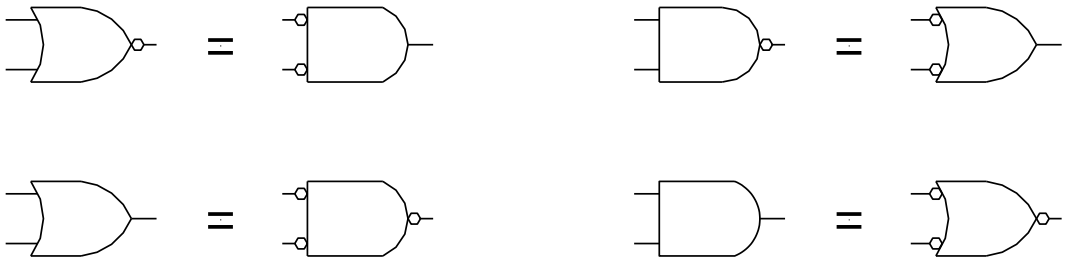
- Developing NAND circuits from K-maps
- Two-level implementations
  - Convert from AND/OR to NAND (again!)
- Multi-level NAND implementations
  - Convert from a network of AND/ORs
- Exclusive OR
  - Comparison with SOP
- Parity checking and detecting circuitry
  - Efficient with XOR gates!

NAND-NAND & NOR-NOR Networks

DeMorgan's Law:

$$(a + b)' = a'\ b' \qquad (a\ b)' = a' + b'$$

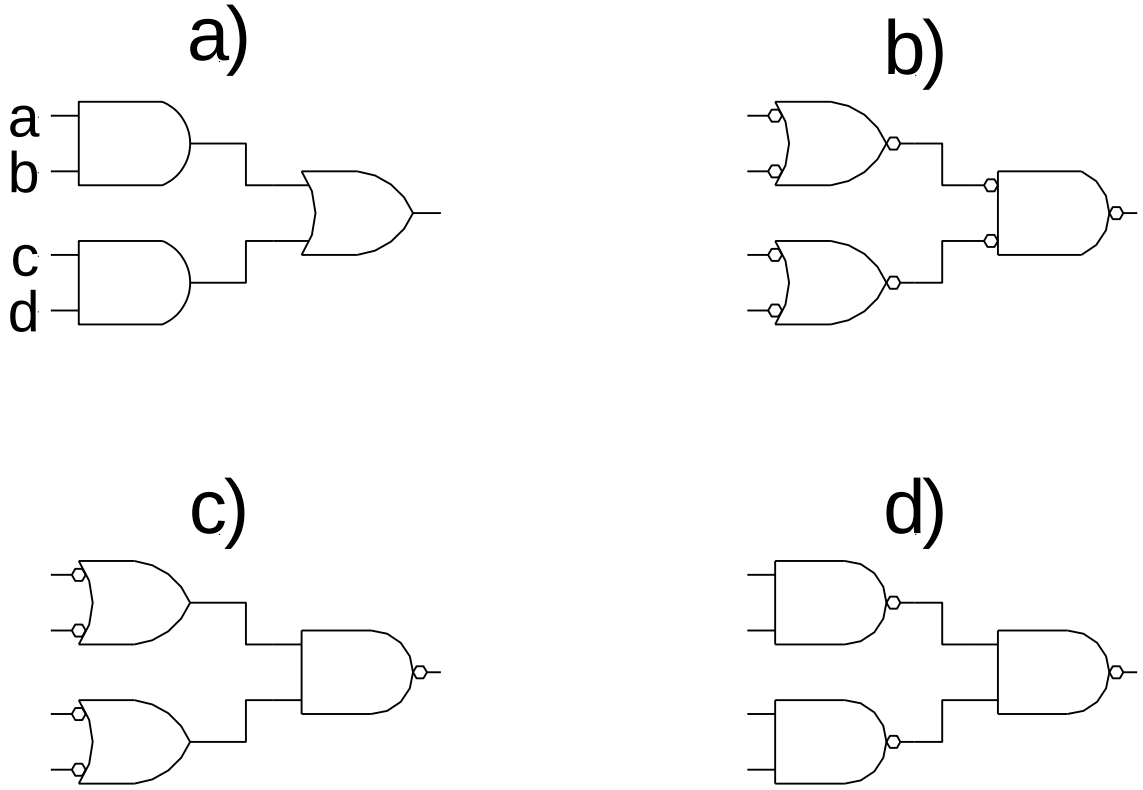
$$a + b = (a'\ b')' \qquad (a\ b) = (a' + b')'$$



push bubbles **or** ***introduce in pairs*** **or** ***remove pairs.***

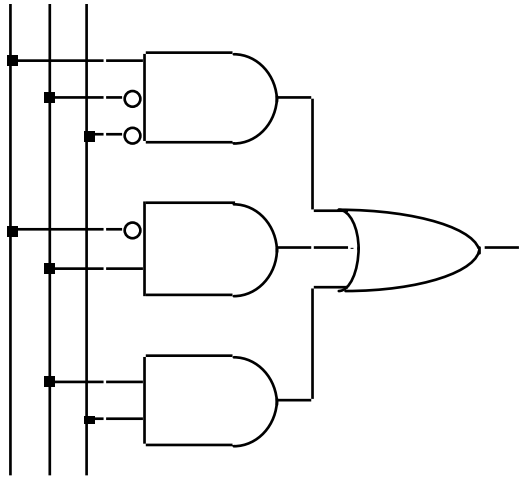
NAND-NAND Networks

- **Mapping from AND/OR to NAND/NAND**

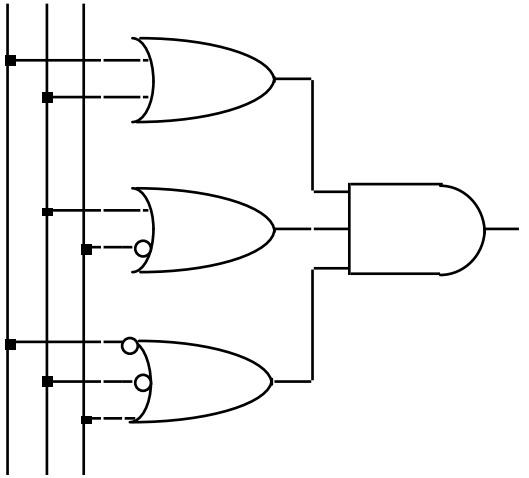


Implementations of Two-level Logic

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

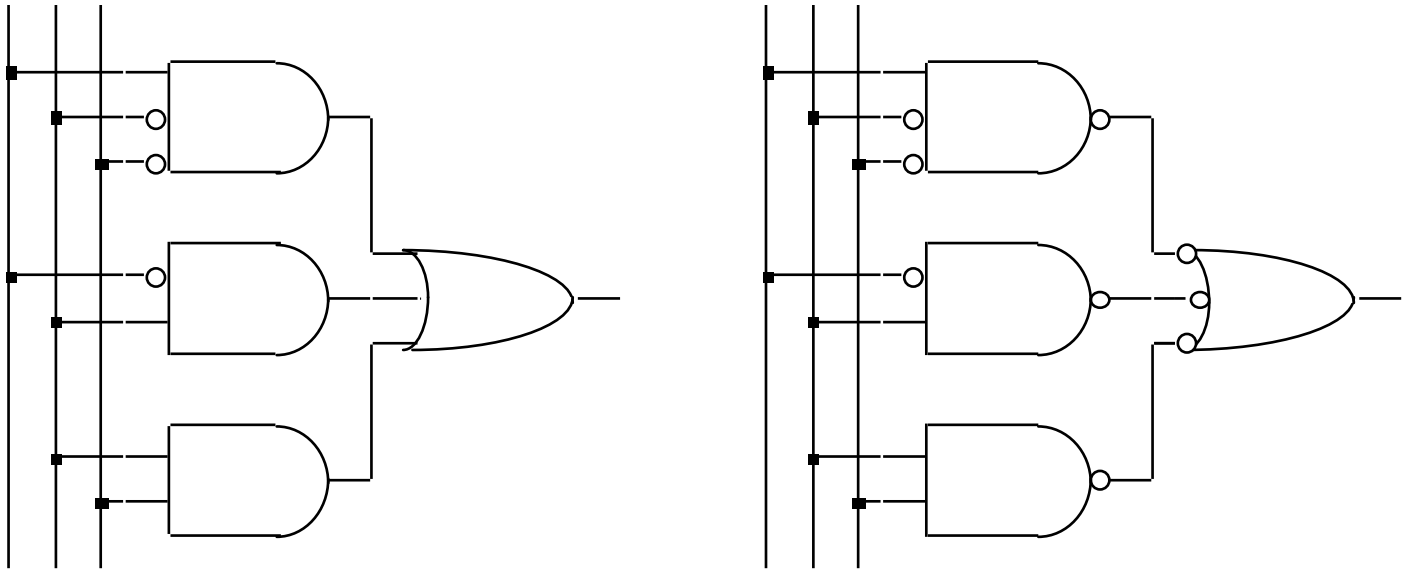


- **Product-of-sums**
  - **OR gates to form sum terms (maxterms)**
  - **AND gates to form product**



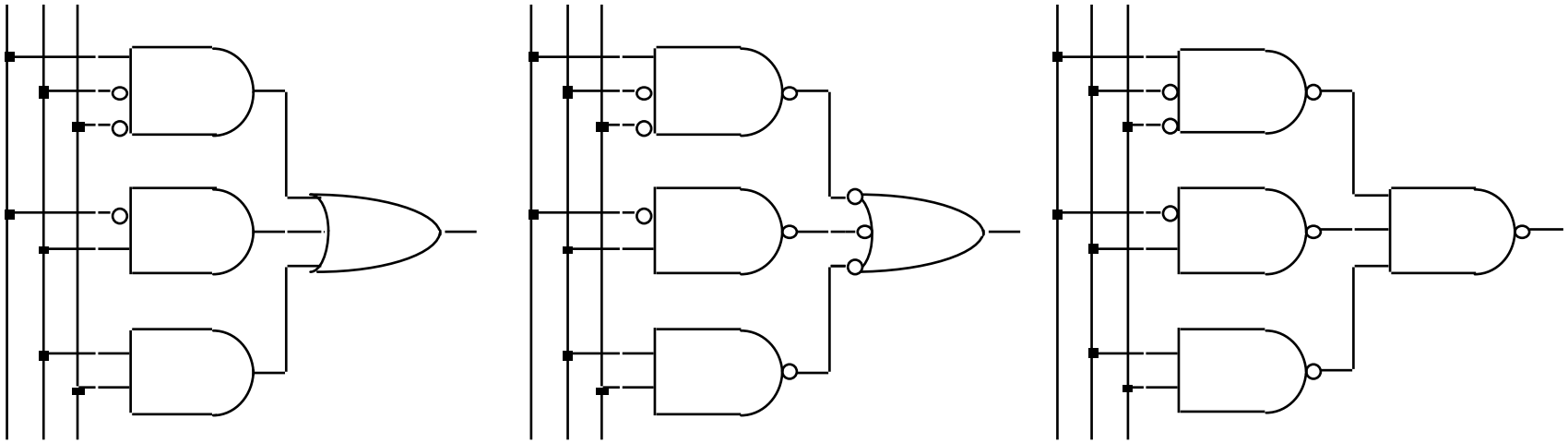
Two-level Logic using NAND Gates

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



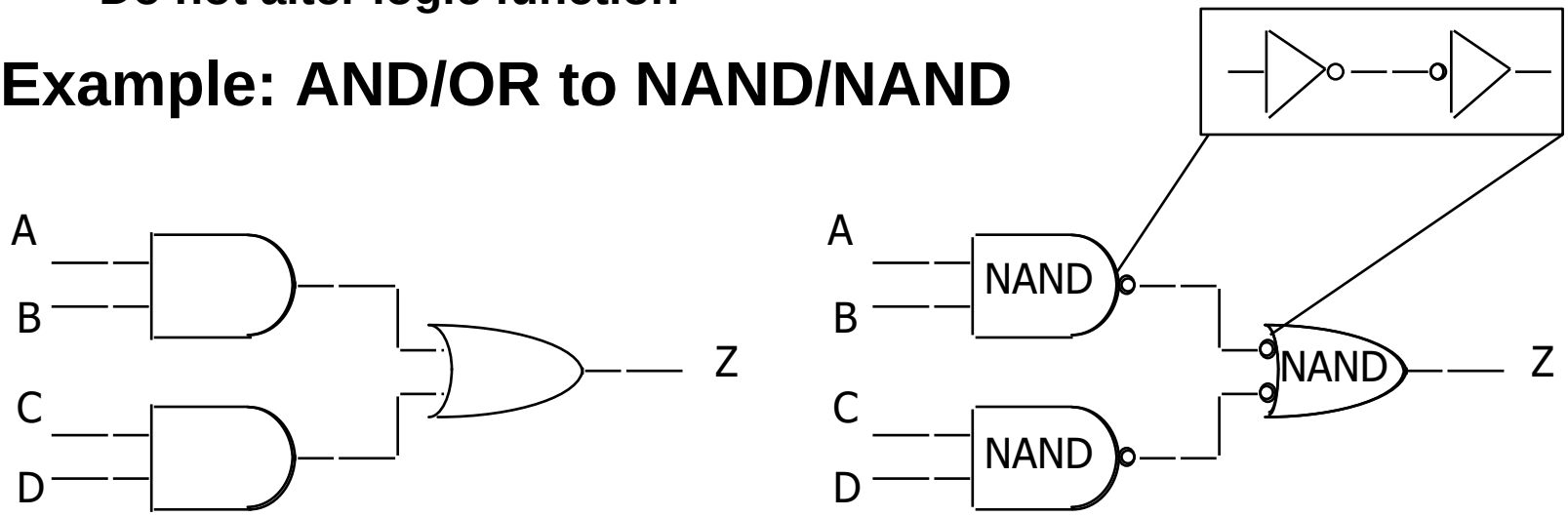
Two-level Logic using NAND Gates (cont'd)

- **OR gate with inverted inputs is a NAND gate**
  - **de Morgan's:** $A' + B' = (A \cdot B)'$
- **Two-level NAND-NAND network**
  - **Inverted inputs are not counted**
  - **In a typical circuit, inversion is done once and signal distributed**



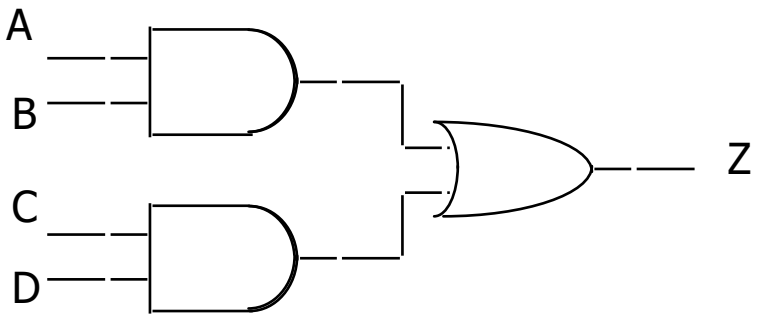
Conversion Between Forms

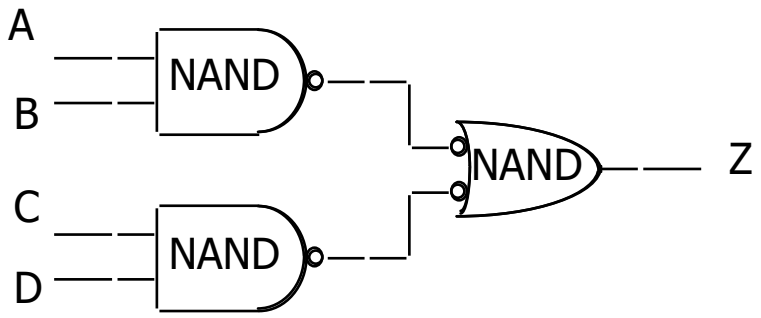
- **Convert from networks of ANDs and ORs to networks of NANDs and NORs**
  - **Introduce appropriate inversions ("bubbles")**
- **Each introduced "bubble" must be matched by a corresponding "bubble"**
  - **Conservation of inversions**
  - **Do not alter logic function**
- **Example: AND/OR to NAND/NAND**



Conversion Between Forms (cont'd)

- **Example: verify equivalence of two forms**

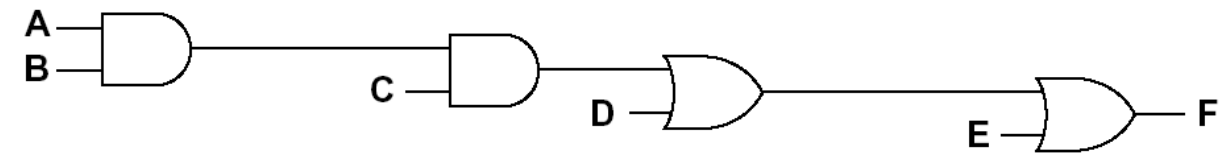




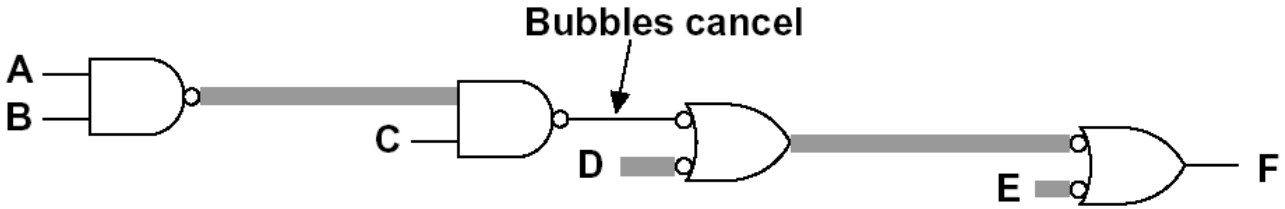
$$
\begin{aligned}
Z &= [\,(A \cdot B)' \cdot (C \cdot D)'\,]' \\
&= [\,(A' + B') \cdot (C' + D')\,]' \\
&= [\,(A' + B')' + (C' + D')'\,] \\
&= (A \cdot B) + (C \cdot D) \checkmark
\end{aligned}
$$

Conversion to NAND Gates

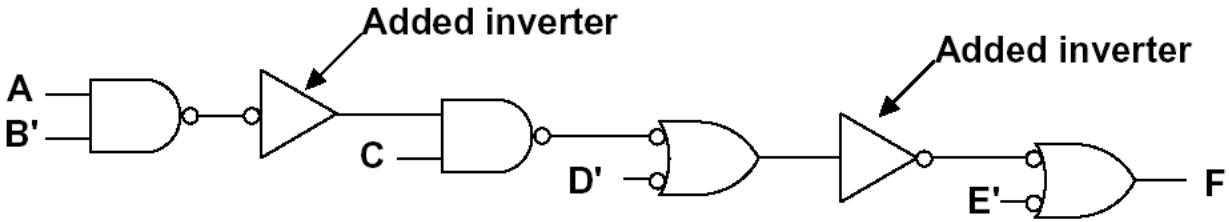
- **Start with SOP (Sum of Products)**
  - **circle 1s in K-maps**
- **Find network of OR and AND gates**



(a) AND_OR network



(b) First step in NAND conversion



(c) Completed conversion

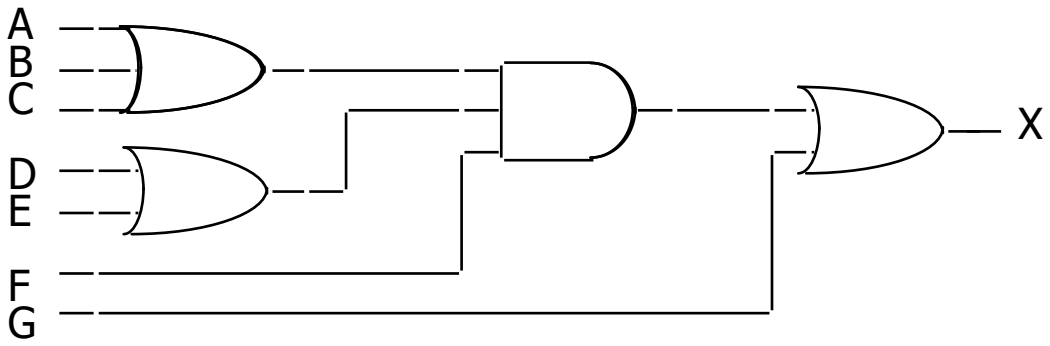
Multi-level Logic

° x = A D F + A E F + B D F + B E F + C D F + C E F + G

- Reduced sum-of-products form – already simplified
- 6 x 3-input AND gates + 1 x 7-input OR gate (may not exist!)
- 25 wires (19 literals plus 6 internal wires)

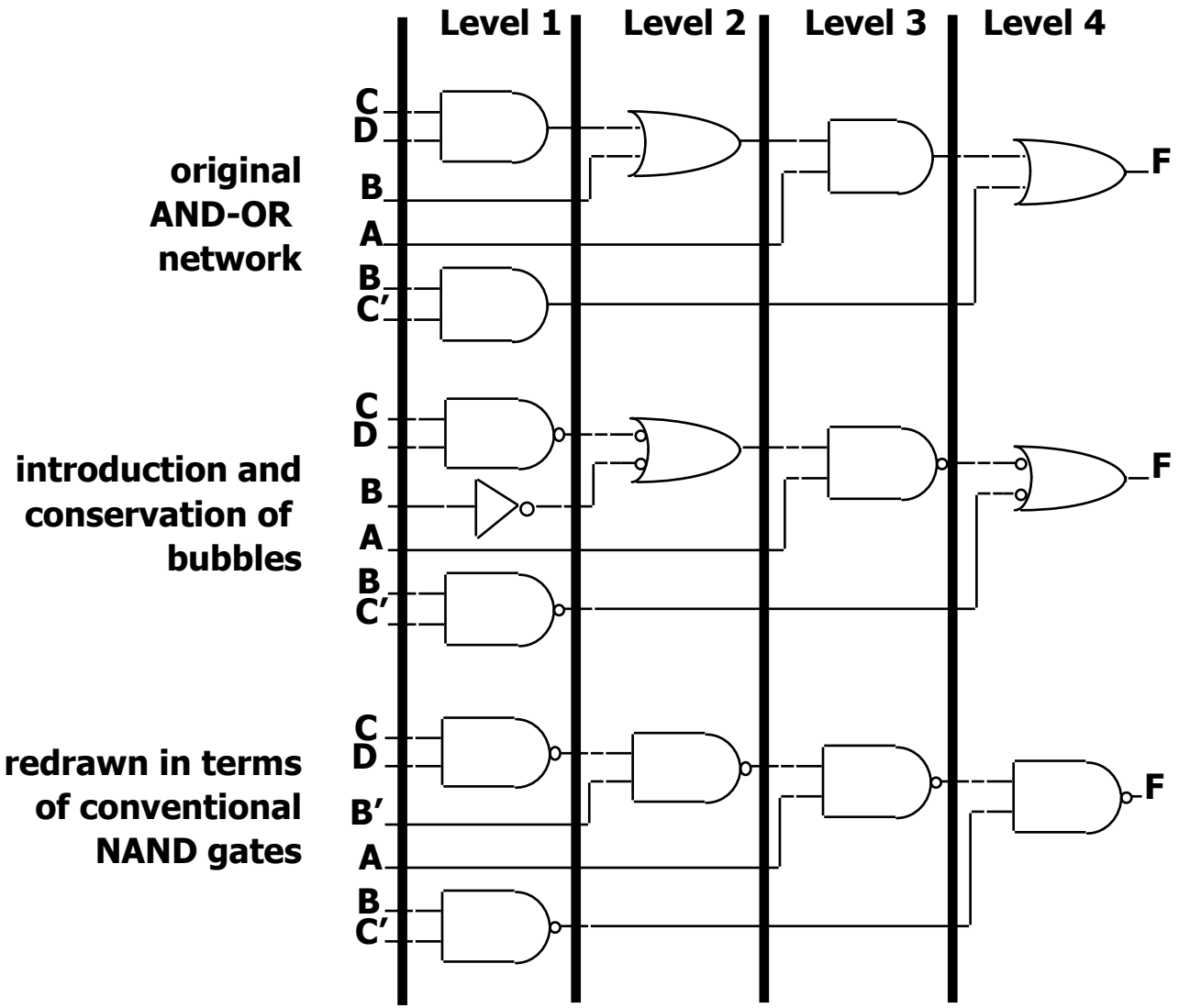
° x = (A + B + C) (D + E) F + G

- Factored form – not written as two-level S-o-P
- 1 x 3-input OR gate, 2 x 2-input OR gates, 1 x 3-input AND gate
- 10 wires (7 literals plus 3 internal wires)



Conversion of Multi-level Logic to NAND Gates

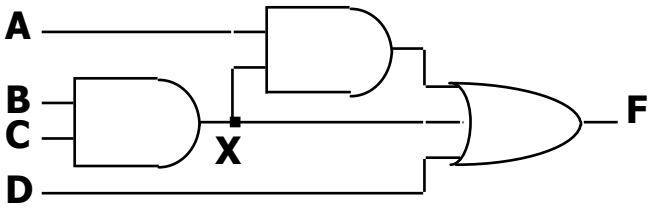
° F = A (B + C D) + B C'



Conversion Between Forms

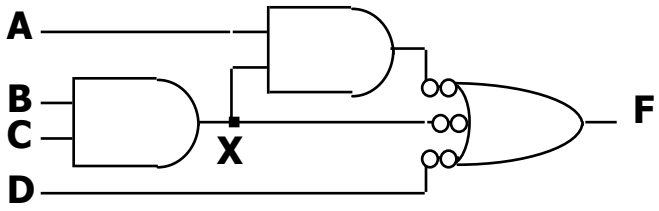
- Example

(a)



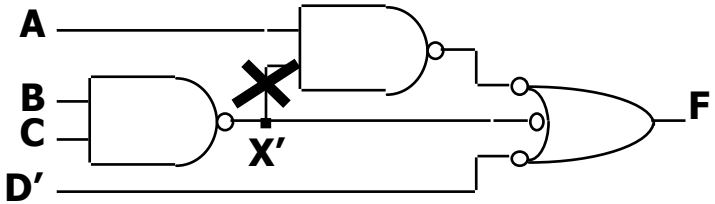
Original circuit

(b)



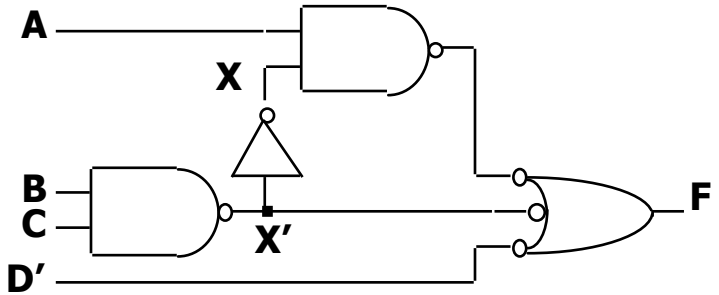
Add double bubbles at inputs

(c)



Distribute bubbles
some mismatches

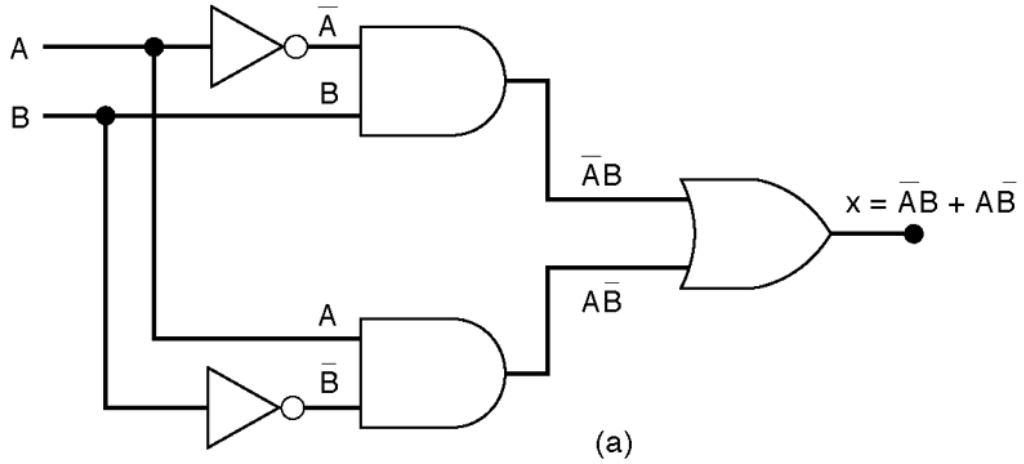
(d)



Insert inverters to fix mismatches

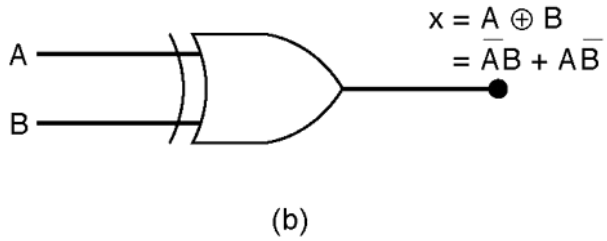
Exclusive-OR and Exclusive-NOR Circuits

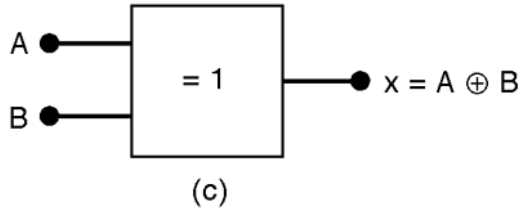
Exclusive-OR (XOR) produces a HIGH output whenever the two inputs are at opposite levels.



A	B	x
0	0	0
0	1	1
1	0	1
1	1	0

XOR gate symbols

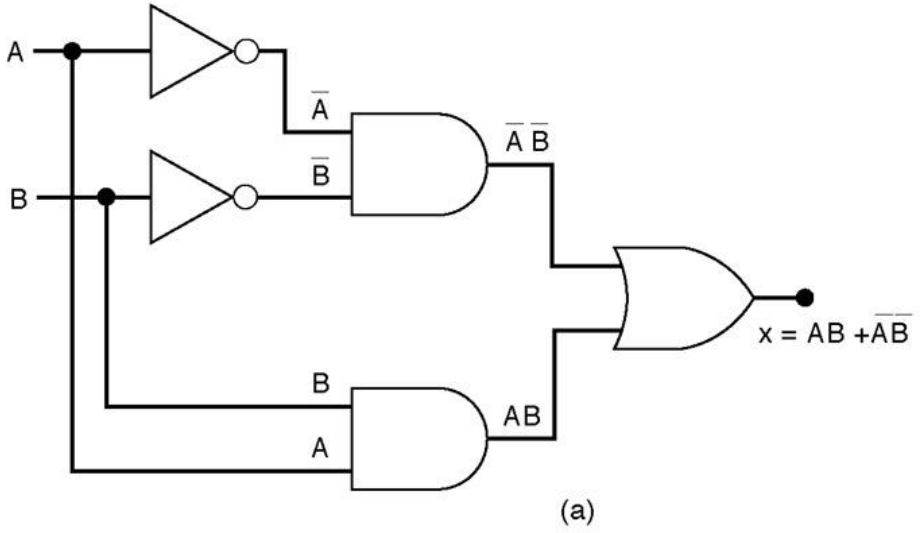




Exclusive-NOR Circuits

Exclusive-NOR (XNOR) :

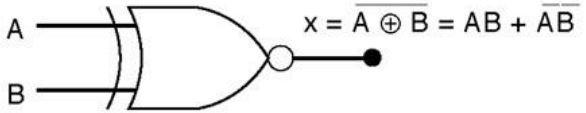
Exclusive-NOR (XNOR) produces a HIGH output whenever the two inputs are at the same level.



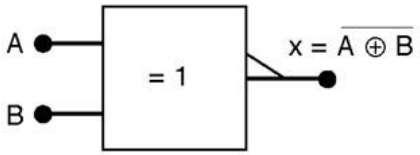
(a)

A	B	x
0	0	1
0	1	0
1	0	0
1	1	1

XNOR gate symbols



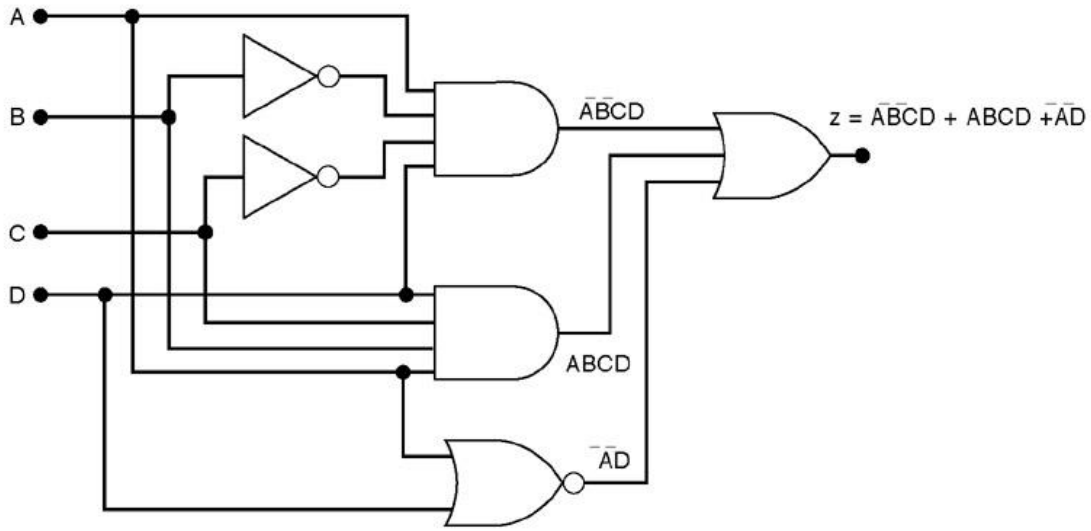
(b)



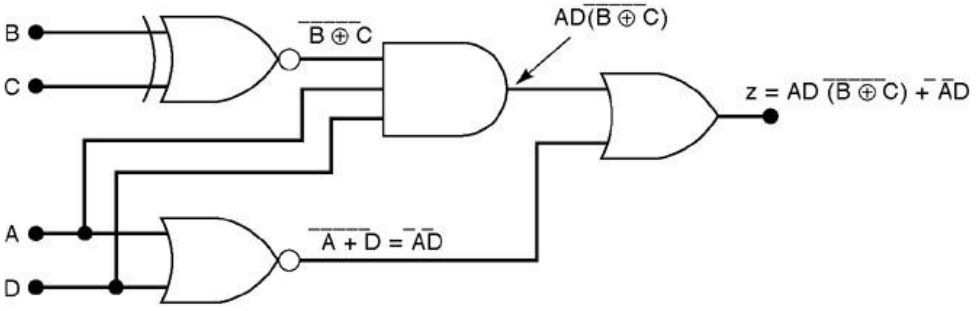
(c)

Exclusive-NOR Circuits

XNOR gate may be used to simplify circuit implementation.



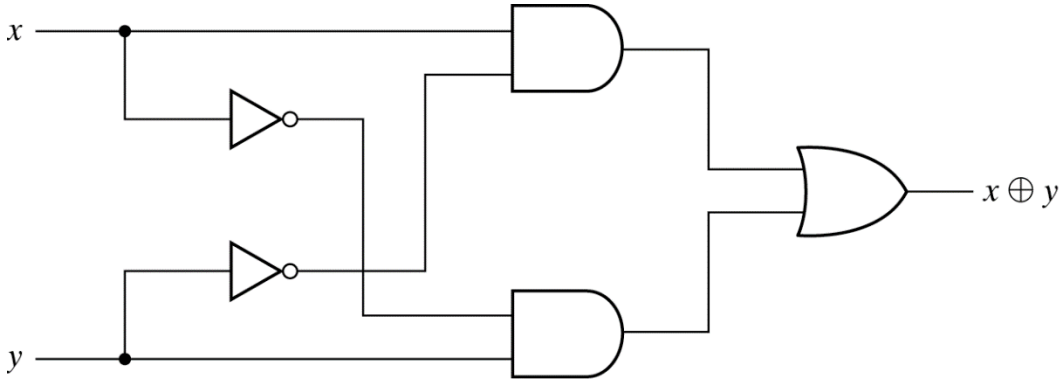
(a)



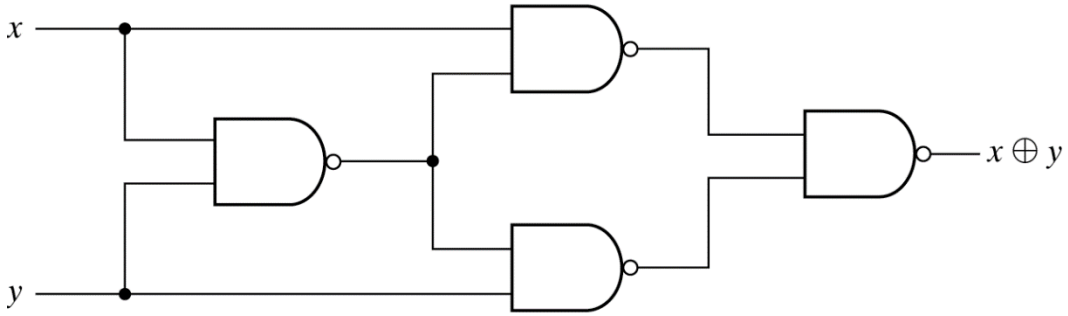
(b)

XOR Function

- **XOR function can also be implemented with AND/OR gates (also NANDs).**



(a) With AND-OR-NOT gates



(b) With NAND gates

Fig. 3-32 Exclusive-OR Implementations

° The first NAND gate performs the operation

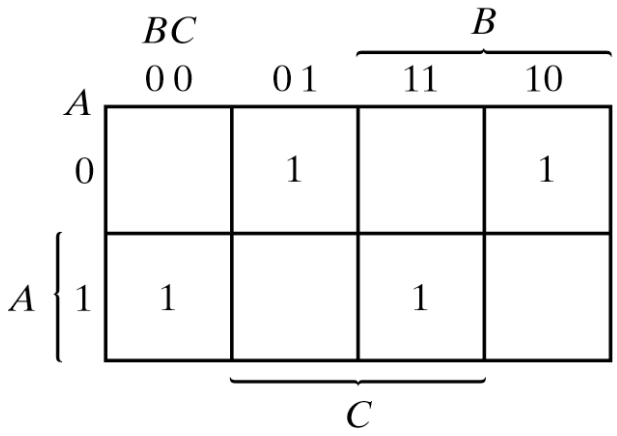
(xy)' = (x' + y').

° The other two-level NAND circuits prcduces the **sum** of products of its inputs:

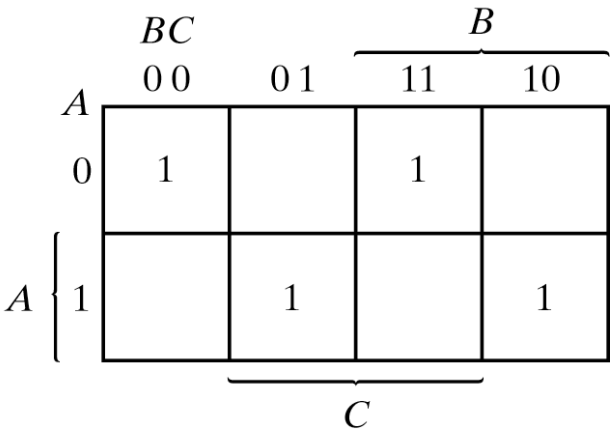
(x' + y')x + *(x'* + y')y = *xy'* + x'y = x XOR **y**

XOR Function

- **Even function – even number of inputs are 1.**
- **Odd function – odd number of inputs are 1.**



(a) Odd function
$F = A \oplus B \oplus C$

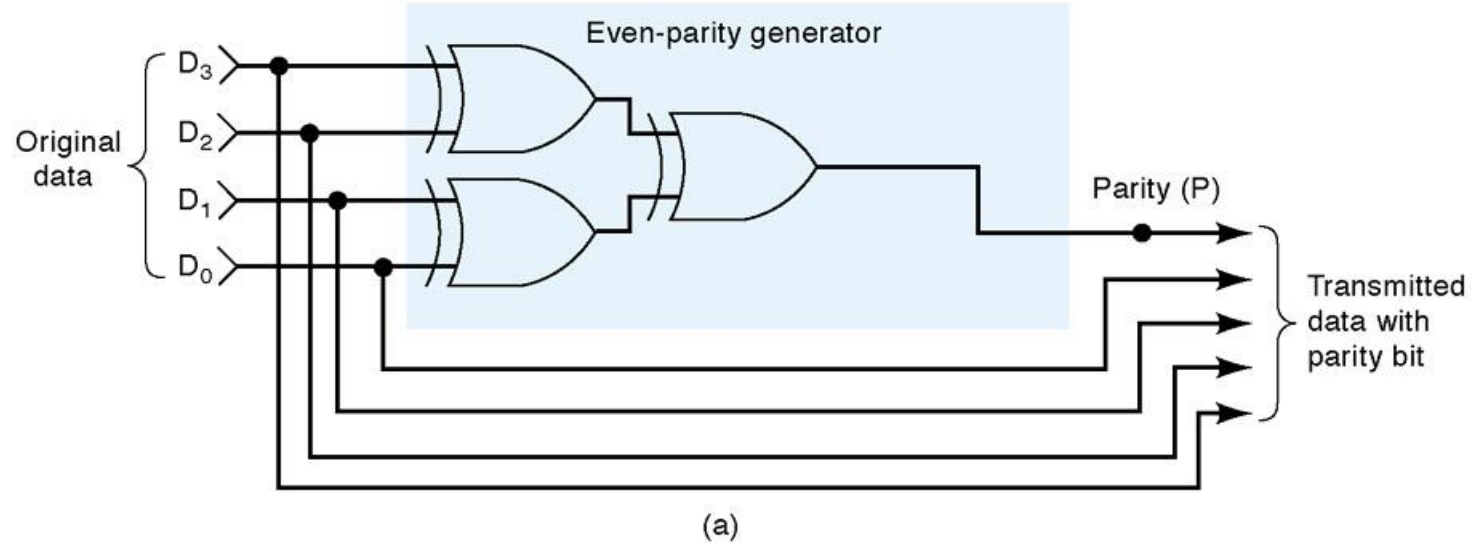


(a) Even function
$F = (A \oplus B \oplus C)'$

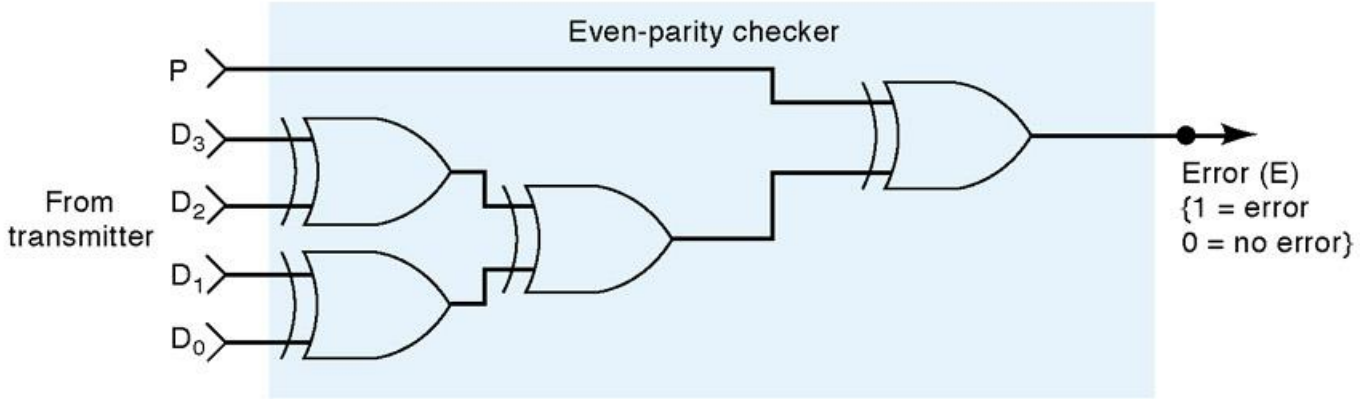
Fig. 3-33 Map for a Three-variable Exclusive-OR Function

Parity Generation and Checking

FIGURE 4-25 XOR gates used to implement the parity generator and the parity checker for an even-parity system.



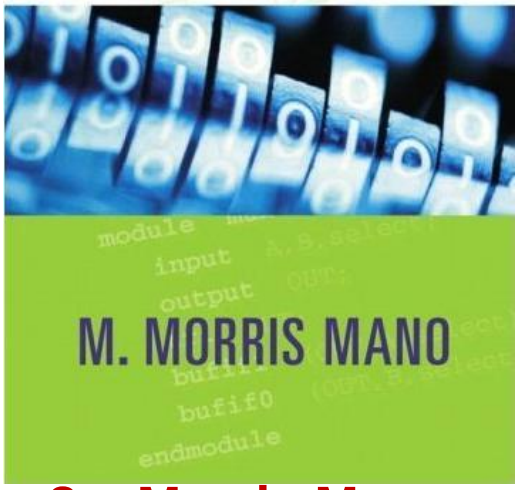
(a)



(b)

Summary

- Follow rules to convert between AND/OR representation and symbols
- Conversions are based on DeMorgan's Law
- NOR gate implementations are also possible
- XORs provide straightforward implementation for some functions
- Used for parity generation and checking
  - XOR circuits could also be implemented using AND/Ors
- Next time: Hazards



Digital Design 3e, Morris Mano

Chapter 4 – Combinational Logic

MODULAR DESIGN OF COMBINATIONAL CIRCUITS

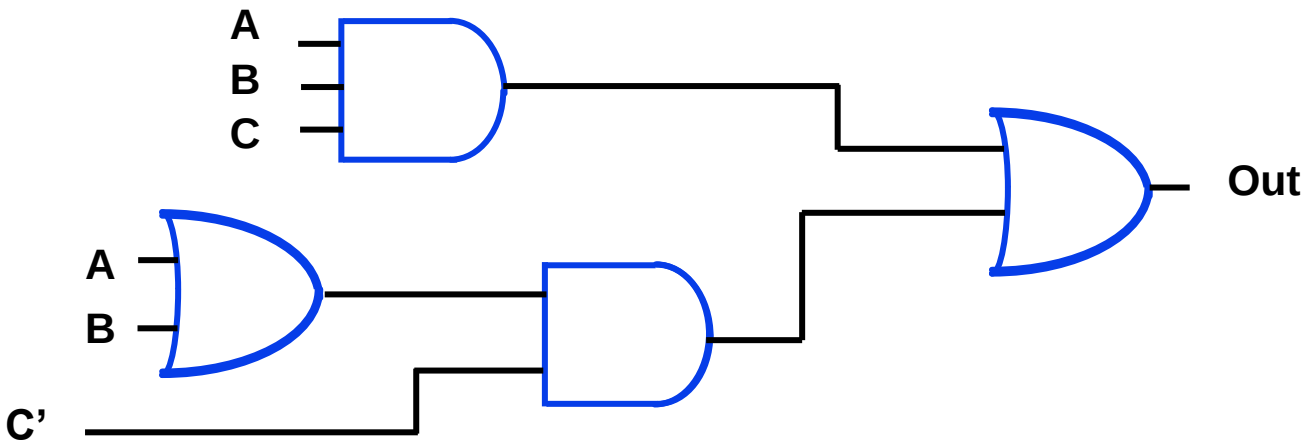
Circuit Analysis Procedure

Overview

- Important concept – analyze digital circuits
  - Given a circuit
    - Create a truth table
    - Create a minimized circuit
- Approaches
  - Boolean expression approach
  - Truth table approach
- Leads to minimized hardware
- Provides insights on how to design hardware
  - Tie in with K-maps (next time)

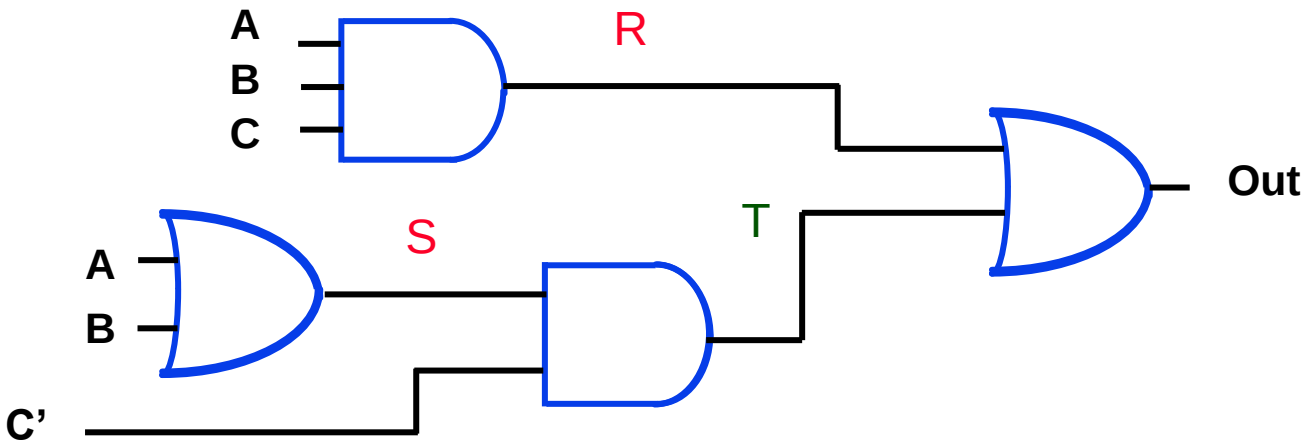
The Problem

- How can we convert from a circuit drawing to an equation or truth table?
- Two approaches
  - Create intermediate equations
  - Create intermediate truth tables



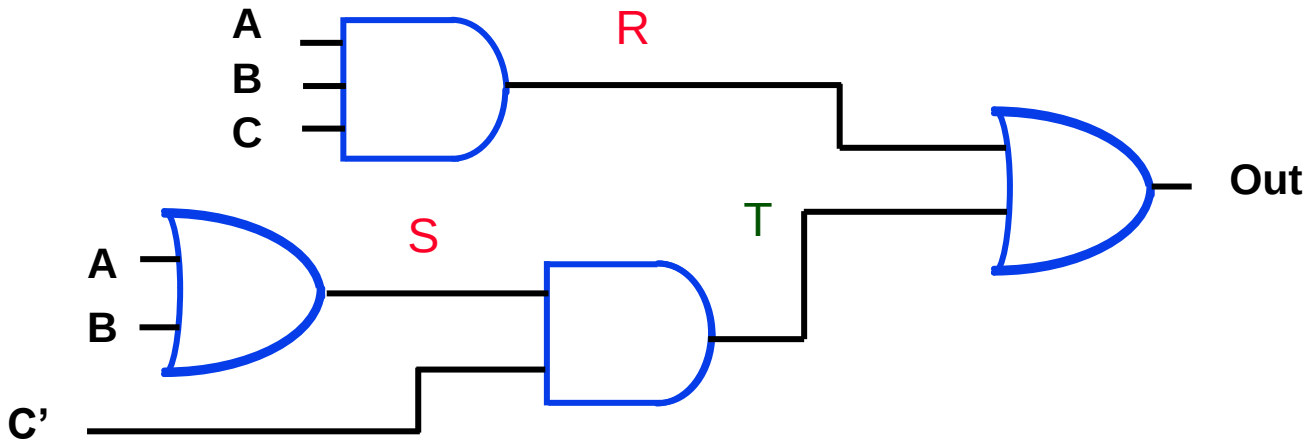
Label Gate Outputs

1. **Label all gate outputs that are a function of input variables.**
2. **Label gates that are a function of input variables and previously labeled gates.**
3. **Repeat process until all outputs are labelled.**



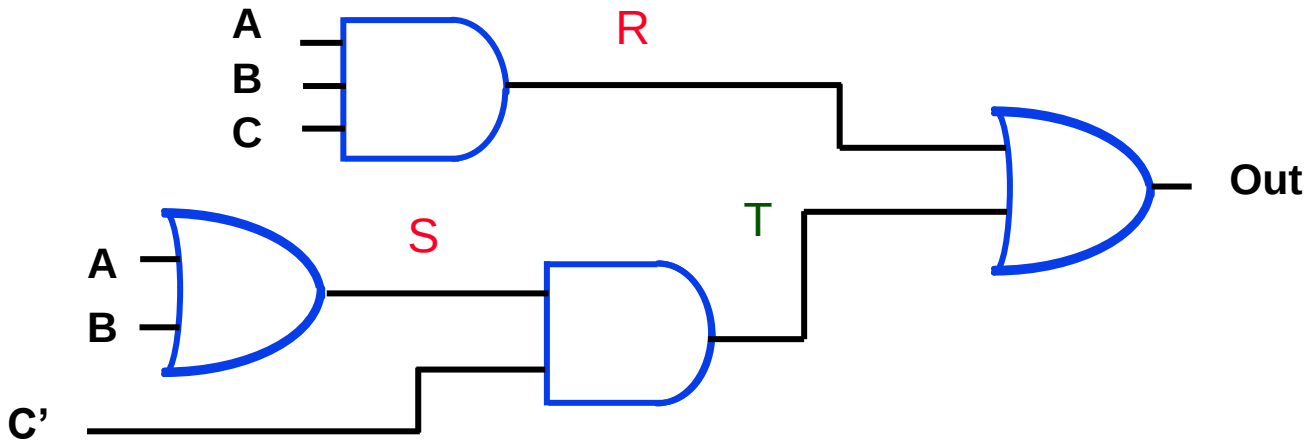
Approach 1: Create Intermediate Equations

- **Step 1: Create an equation for each gate output based on its input.**
  - **R = ABC**
  - **S = A + B**
  - **T = C'S**
  - **Out = R + T**



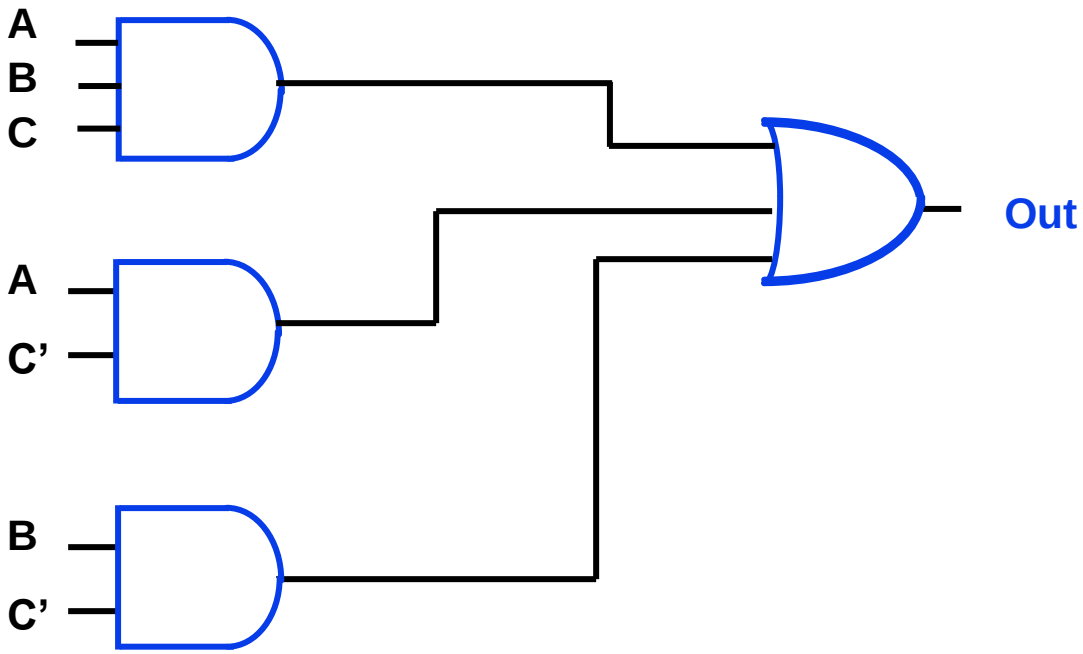
Approach 1: Substitute in subexpressions

- **Step 2: Form a relationship based on input variables (A, B, C)**
  - **R = ABC**
  - **S = A + B**
  - **T = C'S = C'(A + B)**
  - **Out = R+T = ABC + C'(A+B)**



Approach 1: Substitute in subexpressions

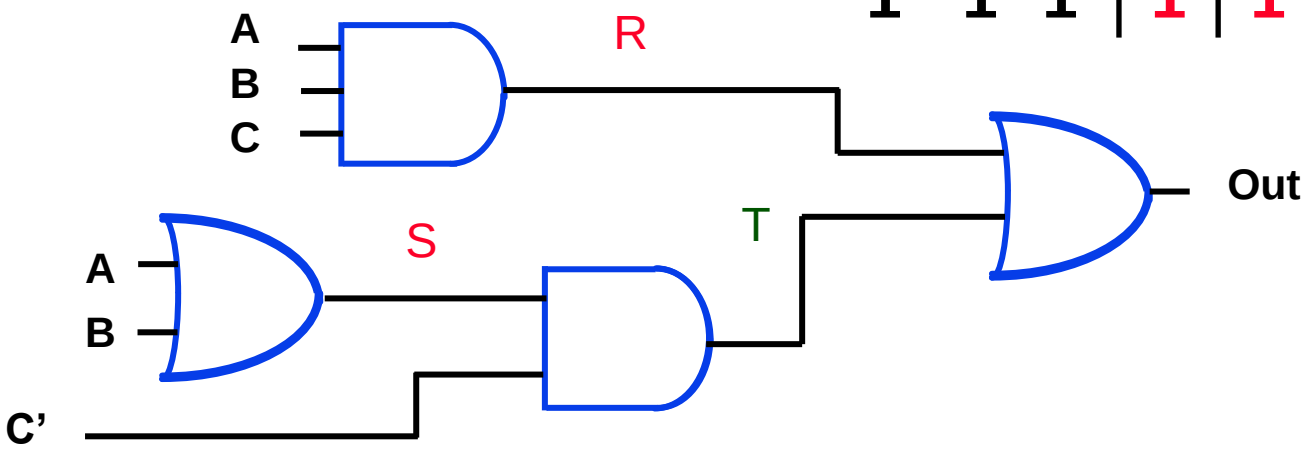
- **Step 3: Expand equation to SOP final result**
  - **Out = ABC + C'(A+B) = ABC + AC' + BC'**



Approach 2: Truth Table

- **Step 1: Determine outputs for functions of input variables.**

A	B	C	R	S
0	0	0	0	0
0	0	1	0	0
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

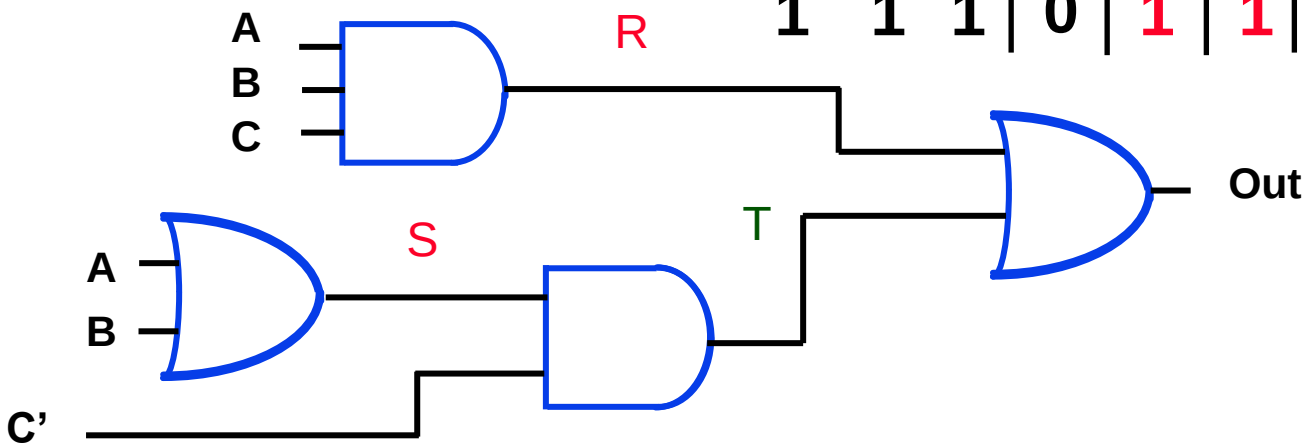


Approach 2: Truth Table

- **Step 2: Determine outputs for functions of intermediate variables.**

T = S * C’

A	B	C	C’	R	S	T
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	1	0	1	0	1	1
0	1	1	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	0	1	0
1	1	0	1	0	1	1
1	1	1	0	1	1	0

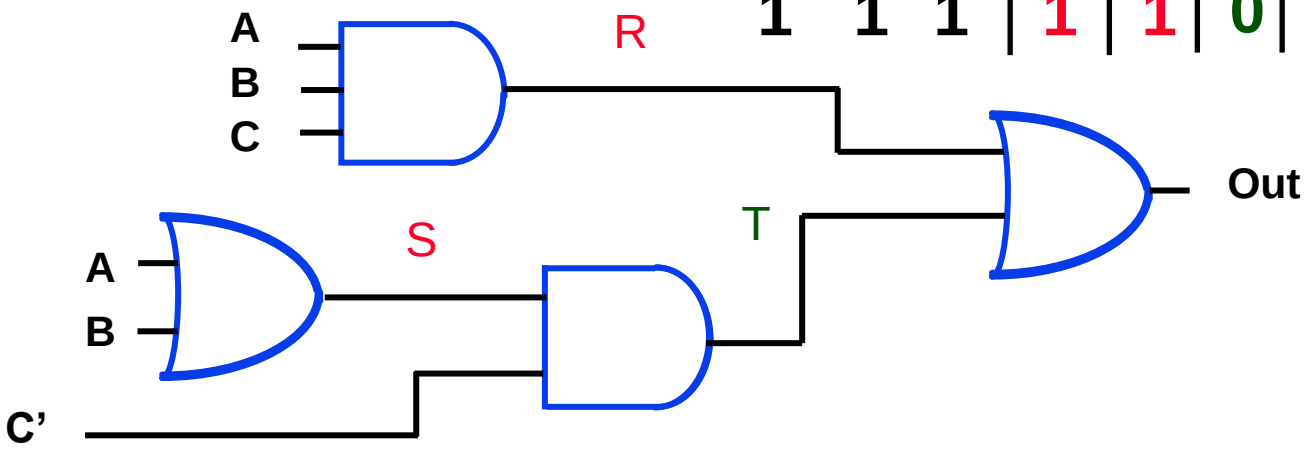


Approach 2: Truth Table

- **Step 3: Determine outputs for function.**

R + T = Out

A	B	C	R	S	T	Out
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	1	1	1
0	1	1	0	1	0	0
1	0	0	0	1	1	1
1	0	1	0	1	0	0
1	1	0	0	1	1	1
1	1	1	1	1	0	1



More Difficult Example

- **Step 3: Note labels on interior nodes**

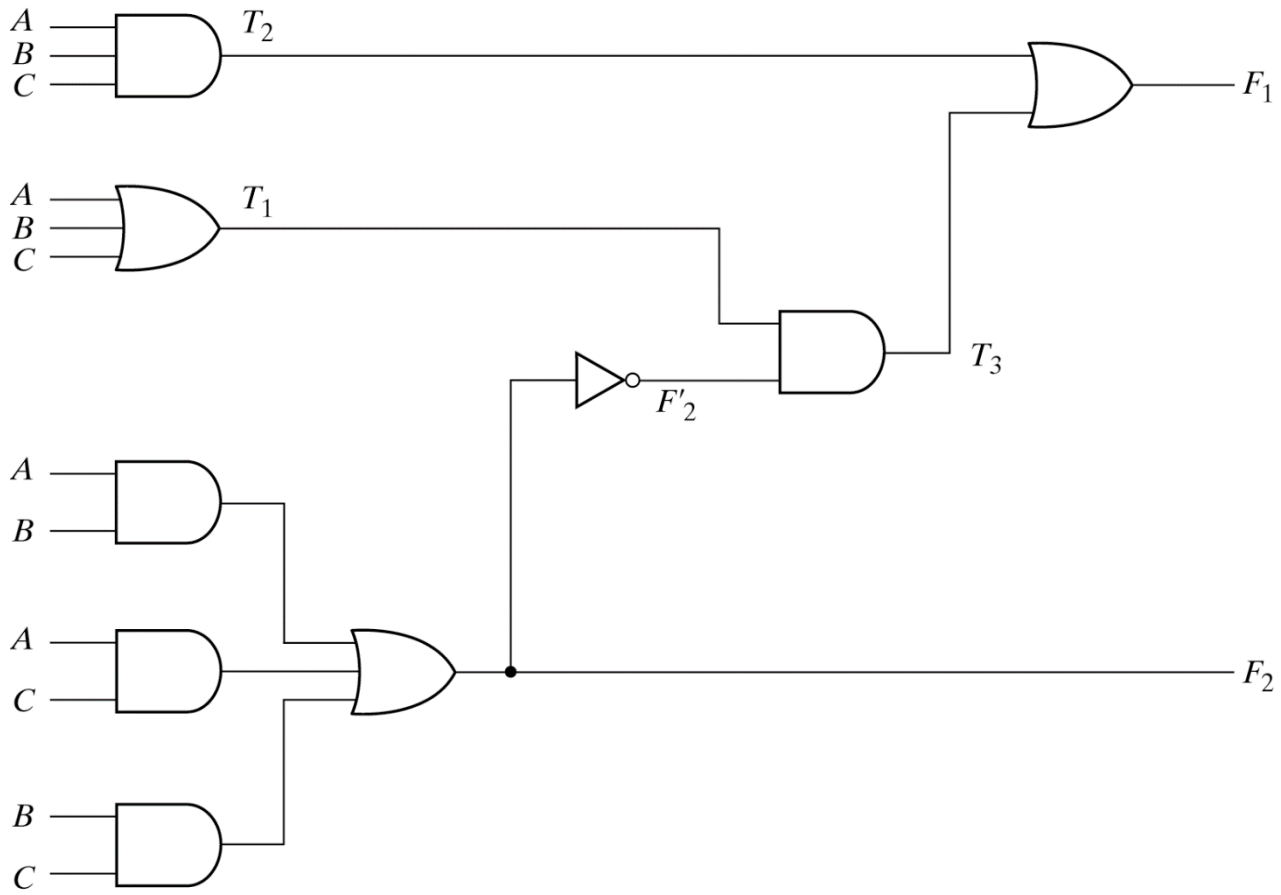


Fig. 4-2 Logic Diagram for Analysis Example

More Difficult Example: Truth Table

- Remember to determine intermediate variables starting from the inputs.
- When all inputs determined for a gate, determine output.
- The truth table can be reduced using K-maps.

A	B	C	F_2	F'_2	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

Summary

- Important to be able to convert circuits into truth table and equation form
  - WHY? ---- leads to minimized sum of product representation
- Two approaches illustrated
  - Approach 1: Create an equation with circuit output dependent on circuit inputs
  - Approach 2: Create a truth table which shows relationship between circuit inputs and circuit outputs
- Both results can then be minimized using K-maps.
- Next time: develop a minimized SOP representation from a high level description

Combinational Design Procedure

Overview

- Design digital circuit from specification
- Digital inputs and outputs known
  - Need to determine logic that can *transform* data
- Start in truth table form
- Create K-map for each output based on function of inputs
- Determine minimized sum-of-product representation
- Draw circuit diagram

Design Procedure (Mano)

Design a circuit from a specification.

1. **Determine number of required inputs and outputs.**
2. **Derive truth table**
3. **Obtain simplified Boolean functions**
4. **Draw logic diagram and verify correctness**

S = A + B + C
R = ABC

A	B	C	R	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Previously, we have learned…

- Boolean algebra can be used to simplify expressions, but not obvious:
  - how to proceed at each step, or
  - if solution reached is minimal.
- Have seen five ways to represent a function:
  - Boolean expression
  - truth table
  - logic circuit
  - minterms/maxterms
  - Karnaugh map

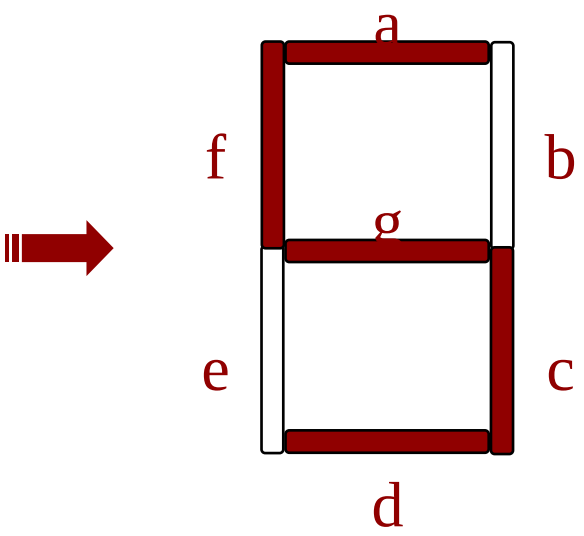
Combinational logic design

- Use multiple representations of logic functions
- Use graphical representation to assist in simplification of function.
- Use concept of “don’t care” conditions.
- Example - encoding BCD to seven segment display.
- Similar to approach used by designers in the field.

BCD to Seven Segment Display

- **Used to display binary coded decimal (BCD) numbers using seven illuminated segments.**
- **BCD uses 0's and 1's to represent decimal digits 0 - 9. Need four bits to represent required 10 digits.**
- **Binary coded decimal (BCD) represents each decimal digit with four bits**

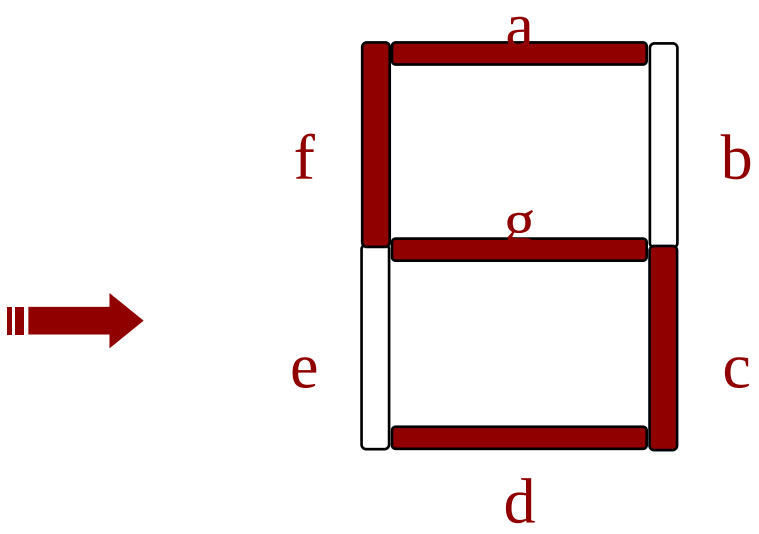
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
.	.	.	.	.
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1



BCD to seven segment display

- **List the segments that should be illuminated for each digit.**

0	a,b,c,d,e,f
1	b,c
2	a,b,d,e,g
3	a,b,c,d,g
4	b,c,f,g
5	a,c,d,f,g
6	a,c,d,e,f,g
7	a,b,c
8	a,b,c,d,e,f,g
9	a,b,c,d,f,g



BCD to seven segment display

- **Derive the truth table for the circuit.**
- **Each output column in one circuit.**

	Inputs				Outputs					
Dec	w	x	y	z	a	b	c	d	e	.
0	0	0	0	0	1	1	1	1	1	.
1	0	0	0	1	0	1	1	0	0	.
2	0	0	1	0	1	1	0	1	1	.
.	.	.	.	.	.	.	.	.	.	.
7	0	1	1	1	1	1	1	0	0	.
8	1	0	0	0	1	1	1	1	1	.
9	1	0	0	1	1	1	1	1	0	.

BCD to seven segment display

- **Find minimal sum-of-products representation for each output**

For segment “a” :

wx \ yz	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11				
10	1	1		

Note: Have only filled in ten squares, corresponding to the ten numerical digits we wish to represent.

Don’t care conditions (BCD display) ...

- Fill in don’t cares for undefined outputs.
  - Note that these combinations of inputs should never happen.
- Leads to a reduced implementation

For segment “a” :

wx \ yz	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	X	X	X	X
10	1	1	X	X

Put in “X” (don’t care), and interpret as either 1 or 0 as desired ….

Don’t care conditions (BCD display) ...

- Circle biggest group of 1’s and Don’t Cares.
- Leads to a reduced implementation

For segment “a” :

wx \ yz	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	X	X	X	X
10	1	1	X	X

$$F_{a1} = y$$

Don’t care conditions (BCD display)

- **Circle biggest group of 1’s and Don’t Cares.**
- **Leads to a reduced implementation**

For segment “a” :

wx \ yz	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	X	X	X	X
10	1	1	X	X

$$F_{a2} = w$$

Don’t care conditions (BCD display) ...

- **Circle biggest group of 1’s and Don’t Cares.**
- **All 1’s should be covered by at least one implicant**

For segment “a” :

wx \ yz	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	X	X	X	X
10	1	1	X	X

$$F_{a3} = \bar{x}\bar{z}$$

wx \ yz	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	X	X	X	X
10	1	1	X	X

$$F_{a4} = xz$$

Don’t care conditions (BCD display) ...

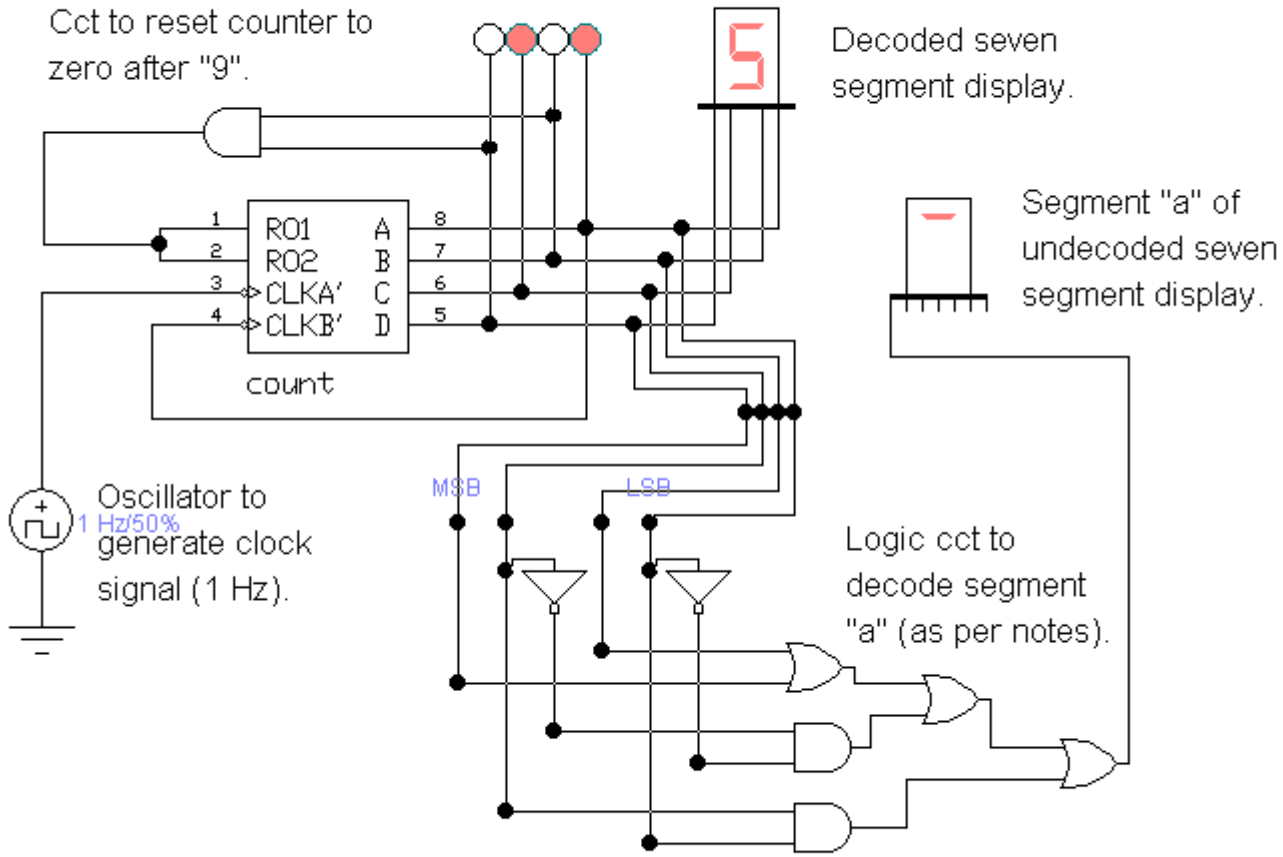
- **Put all the terms together**
- **Generate the circuit**

For segment “a” :

wx \ yz	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	X	X	X	X
10	1	1	X	X

$$F = y + w + \bar{x}\bar{z} + xz$$

Example of seven segment display decoding.



Hint: Select a component and then push "?" from main menu bar to get info on what that component does and how it works.

BCD to seven segment display

- **Derive the truth table for the circuit.**
- **Each output column in one circuit.**

	Inputs				Outputs					
Dec	w	x	y	z	a	b	c	d	e	.
0	0	0	0	0	1	1	1	1	1	.
1	0	0	0	1	0	1	1	0	0	.
2	0	0	1	0	1	1	0	1	1	.
.	.	.	.	.	.	.	.	.	.	.
7	0	1	1	1	1	1	1	0	0	.
8	1	0	0	0	1	1	1	1	1	.
9	1	0	0	1	1	1	1	1	0	.

BCD to seven segment display

- **Find minimal sum-of-products representation for each output**

For segment "b" :

wx \ yz	00	01	11	10
00	1	1	1	1
01	1	0	1	0
11				
10	1	1		

See if you complete this example.

Summary

- **Need to formulate circuits from problem descriptions**
  1. **Determine number of inputs and outputs**
  2. **Determine truth table format**
  3. **Determine K-map**
  4. **Determine minimal SOP**
- **There may be multiple outputs per design**
  - **Solve each output separately**
- **Current approach doesn't have memory.**
  - **This will be covered next week.**