
Module 3

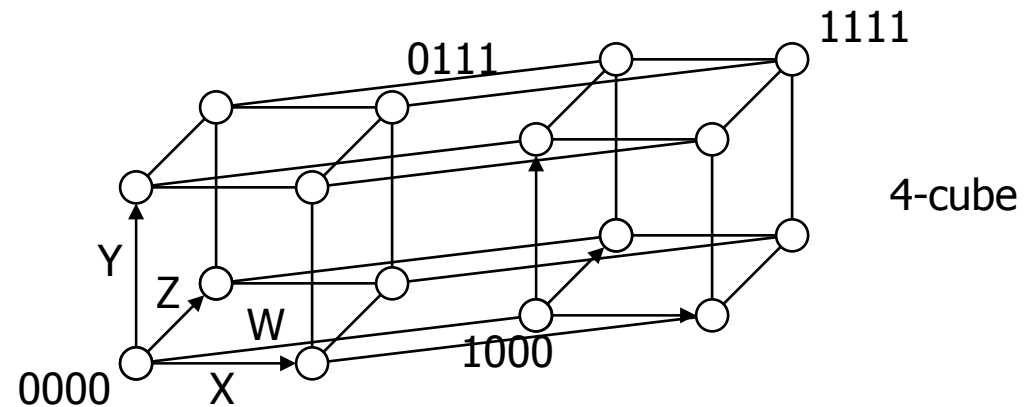
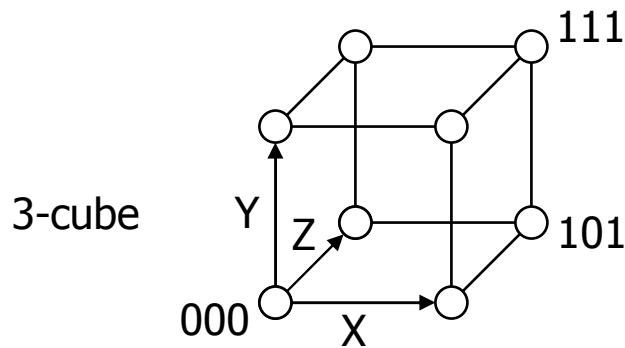
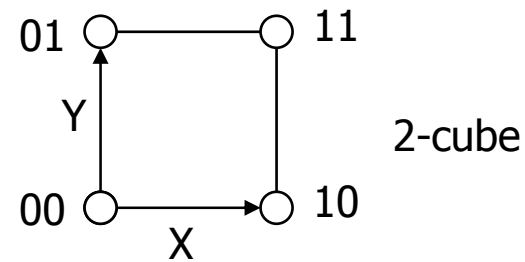
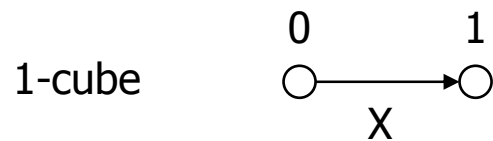
Working with Combination Logic: Supplemental Slides

Louis G. Johnson
School of Electrical and Computer Engineering
Oklahoma State University

Spring 2007

Boolean cubes

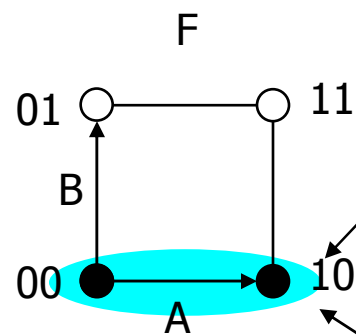
- Visual technique for identifying when the uniting theorem can be applied
- n input variables = n -dimensional "cube"



Mapping truth tables onto Boolean cubes

- Uniting theorem combines two "faces" of a cube into a larger "face"
- Example:

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



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

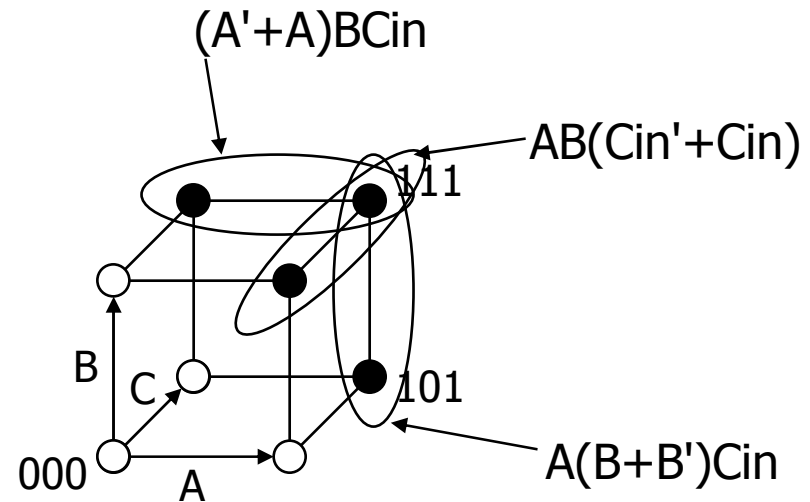
A varies within face, B does not
this face represents the literal B'

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

Three variable example

- Binary full-adder carry-out logic

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

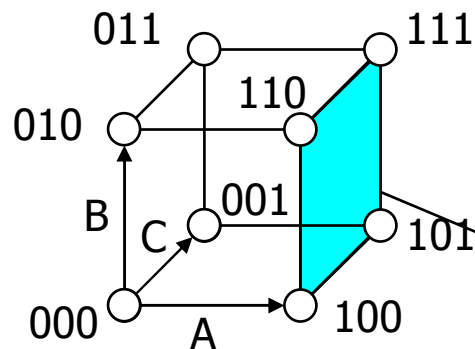


the on-set is completely covered by the combination (OR) of the subcubes of lower dimensionality - note that "111" is covered three times

$$Cout = BCin + AB + ACin$$

Higher dimensional cubes

- Sub-cubes of higher dimension than 2



$$F(A,B,C) = \sum m(4,5,6,7)$$

on-set forms a square
i.e., a cube of dimension 2

*represents an expression in one variable
i.e., 3 dimensions – 2 dimensions*

A is asserted (true) and unchanged
B and C vary

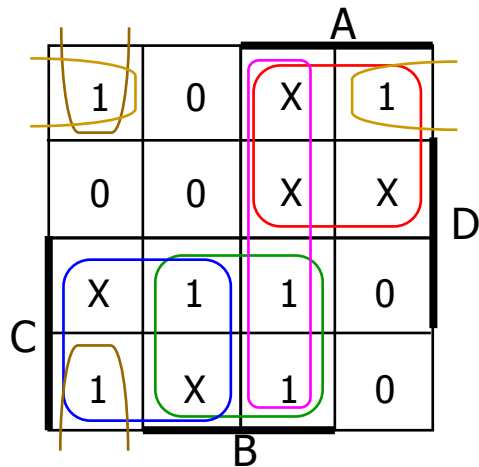
This subcube represents the
literal A

m-dimensional cubes in a n-dimensional Boolean space

- In a 3-cube (three variables):
 - a 0-cube, i.e., a single node, yields a term in 3 literals
 - a 1-cube, i.e., a line of two nodes, yields a term in 2 literals
 - a 2-cube, i.e., a plane of four nodes, yields a term in 1 literal
 - a 3-cube, i.e., a cube of eight nodes, yields a constant term "1"
- In general,
 - an m-subcube within an n-cube ($m < n$) yields a term with $n - m$ literals

Activity

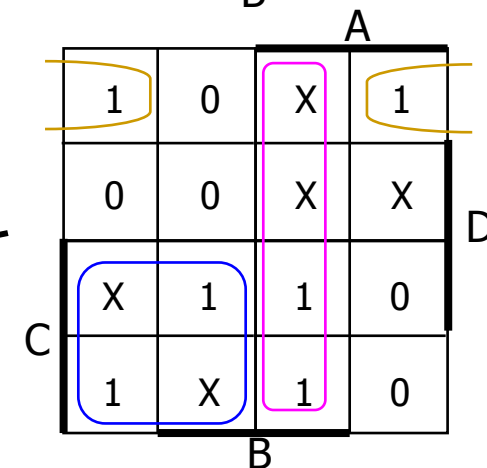
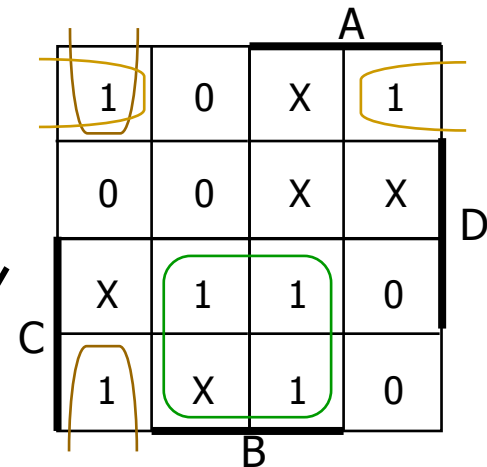
- Minimize the function $F = \Sigma m(0, 2, 7, 8, 14, 15) + d(3, 6, 9, 12, 13)$



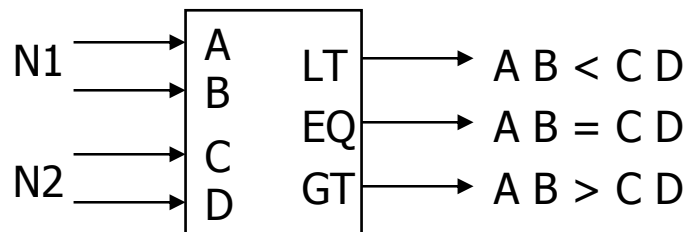
$$F = \cancel{AC'} + \cancel{A'C} + BC + \cancel{AB} + A'B'D' + B'C'D'$$

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

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



Design example: two-bit comparator



block diagram
and
truth table

A	B	C	D	LT	EQ	GT
0	0	0	0	0	1	0
		0	1	1	0	0
		1	0	1	0	0
		1	1	1	0	0
0	1	0	0	0	0	1
		0	1	0	1	0
		1	0	1	0	0
		1	1	1	0	0
1	0	0	0	0	0	1
		0	1	0	0	1
		1	0	0	1	0
		1	1	1	0	0
1	1	0	0	0	0	1
		0	1	0	0	1
		1	0	0	0	1
		1	1	0	1	0

we'll need a 4-variable Karnaugh map
for each of the 3 output functions

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

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

K-map for LT

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

K-map for EQ

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

K-map for GT

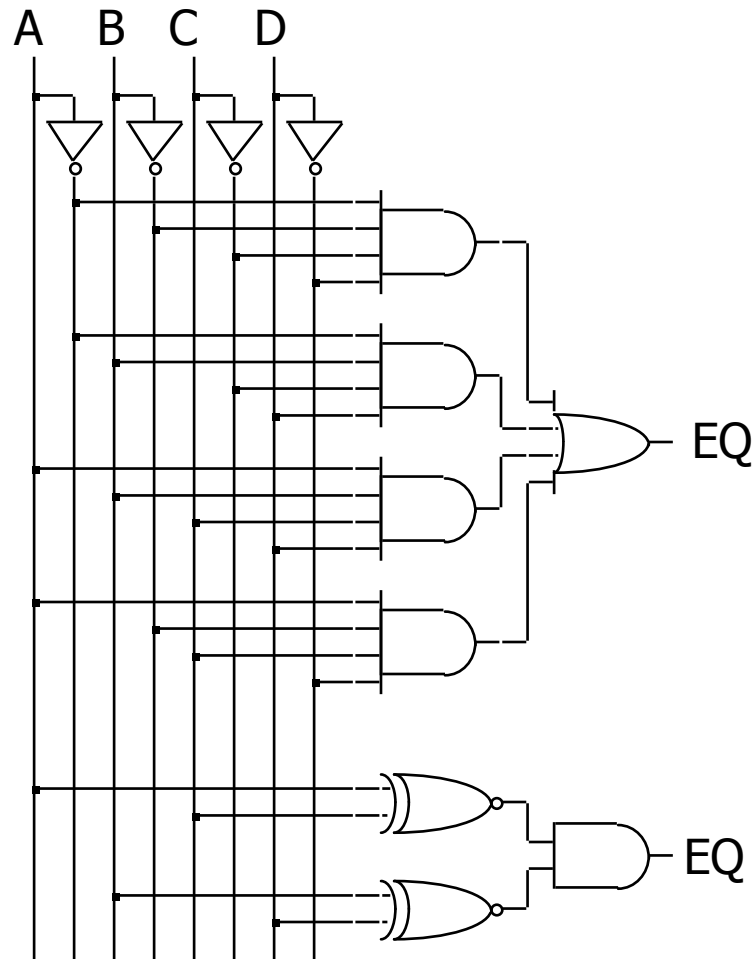
$$LT = A' B' D + A' C + B' C D$$

$$EQ = A' B' C' D' + A' B C' D + A B C D + A B' C D' = (A \text{ xnor } C) \cdot (B \text{ xnor } D)$$

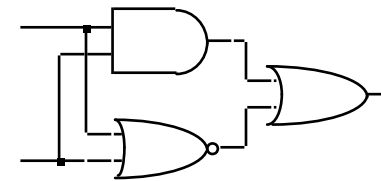
$$GT = B C' D' + A C' + A B D'$$

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

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

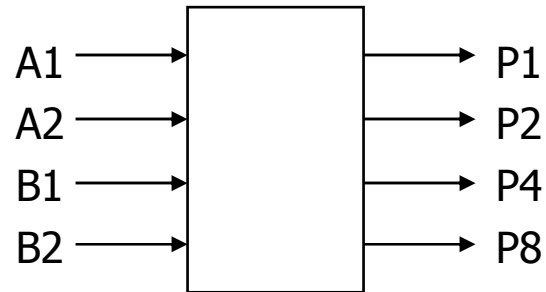


two alternative implementations of EQ with and without XOR



XNOR is implemented with at least 3 simple gates

Design example: 2x2-bit multiplier

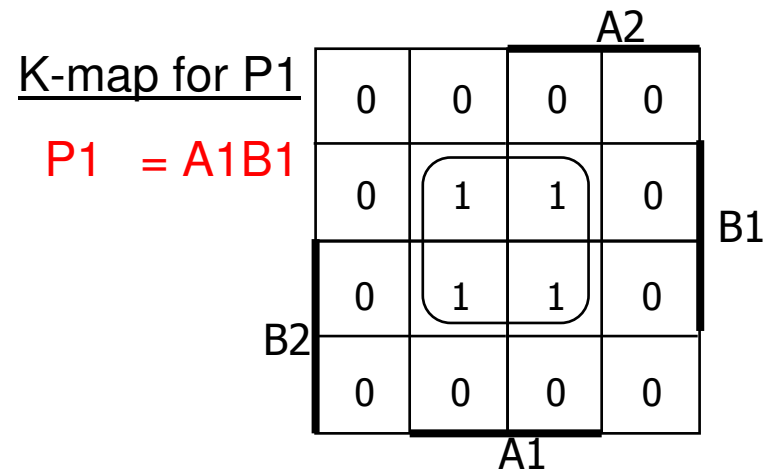
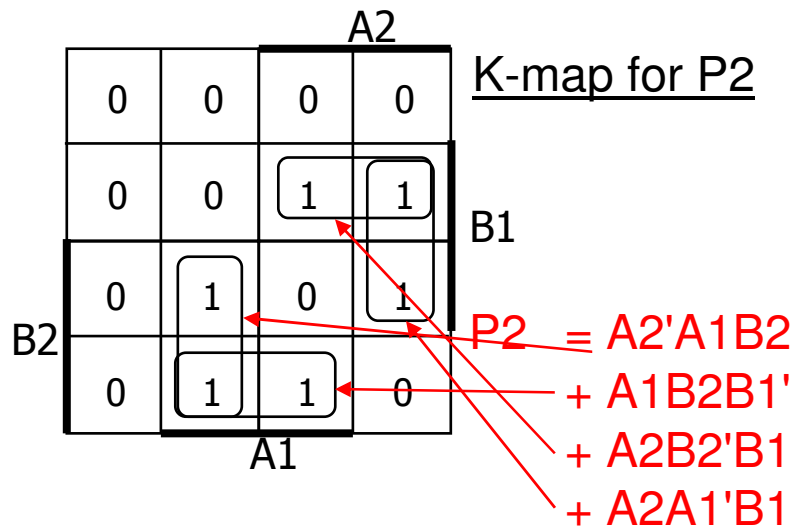
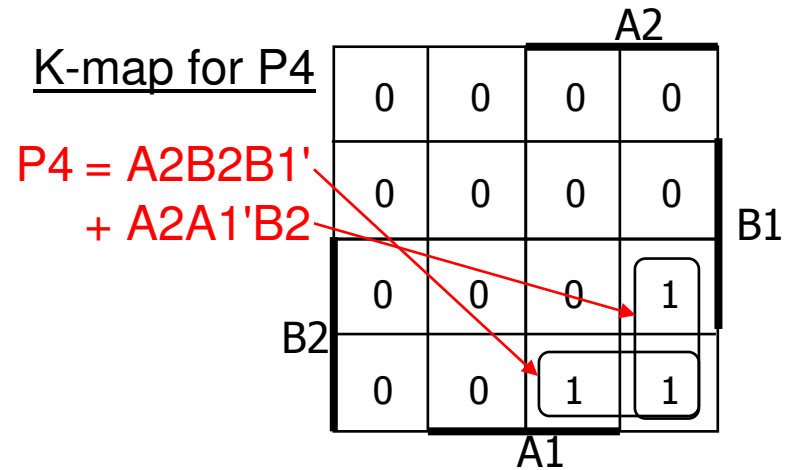
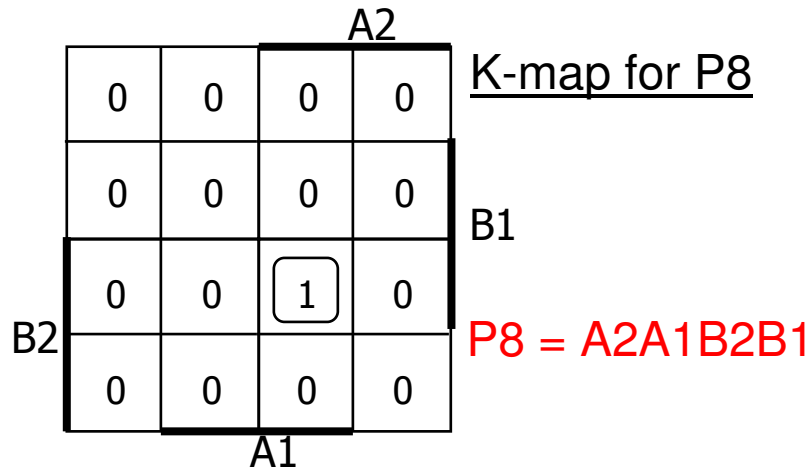


block diagram
and
truth table

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

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

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



Algorithm for two-level simplification (example)

		A		
	X	1	0	1
	0	1	1	1
	0	X	X	0
C	0	1	0	1
		B		

		A		
	X	1	0	1
	0	1	1	1
	0	X	X	0
C	0	1	0	1
		B		

2 primes around $A'BC'D'$

		A		
	X	1	0	1
	0	1	1	1
	0	X	X	0
C	0	1	0	1
		B		

2 primes around $ABC'D$

		A		
	X	1	0	1
	0	1	1	1
	0	X	X	0
C	0	1	0	1
		B		

3 primes around $AB'C'D'$

		A		
	X	1	0	1
	0	1	1	1
	0	X	X	0
C	0	1	0	1
		B		

2 essential primes

		A		
	X	1	0	1
	0	1	1	1
	0	X	X	0
C	0	1	0	1
		B		

minimum cover (3 primes)

Activity

- List all prime implicants for the following K-map:

		A		
	X	0	X	0
	0	1	X	1
	0	X	X	0
C	X	1	1	1
		B		

D

- Which are essential prime implicants?
- What is the minimum cover?

Quine-McCluskey algorithm

- This minimization process can be made into a program, using appropriate algorithms and data structures.
 - Guaranteed to find a “minimal” solution
- Required computation has exponential complexity (run time and storage)-- works well for functions with up to 8-12 variables, but quickly blows up for larger problems.
- Other nearly-optimal methods can be used for larger problems, and usually give minimal results.

CAD Tools for Simplification

Quine-McCluskey Algorithm

Tabular method to systematically find all prime implicants

$$f(A,B,C,D) = \Sigma m(4,5,6,8,9,10,13) + d(0,7,15)$$

Stage 1: Find all prime implicants

Step 1: Fill Column 1 with ON-set and DC-set minterm indices. Group by number of 1's.

Implication Table	
Column I	
0000	
0100	
1000	
0101	
0110	
1001	
1010	
0111	
1101	
1111	

CAD Tools for Simplification

Tabular method to systematically find all prime implicants

$$f(A,B,C,D) = \Sigma m(4,5,6,8,9,10,13) + d(0,7,15)$$

Stage 1: Find all prime implicants

Step 1: Fill Column 1 with ON-set and DC-set minterm indices. Group by number of 1's.

Step 2: Apply Uniting Theorem—
Compare elements of group w/
N 1's against those with N+1 1's.
Differ by one bit implies adjacent.
Eliminate variable and place in
next column.

E.g., 0000 vs. 0100 yields 0-00
0000 vs. 1000 yields -000

When used in a combination,
mark with a check. If cannot be
combined, mark with a star. These
are the prime implicants.

Repeat until no further combinations can be made.

Implication Table		
Column I	Column II	
0000	0-00 -000	
0100 1000	010- 01-0	
0101 0110 1001 1010	100- 10-0 01-1 -101	
0111 1101	011- 1-01	
1111	-111 11-1	

CAD Tools for Simplification

Tabular method to systematically find all prime implicants

$$f(A,B,C,D) = \Sigma m(4,5,6,8,9,10,13) + d(0,7,15)$$

Stage 1: Find all prime implicants

Step 1: Fill Column 1 with ON-set and DC-set minterm indices. Group by number of 1's.

Step 2: Apply Uniting Theorem—
Compare elements of group w/ N 1's against those with N+1 1's. Differ by one bit implies adjacent. Eliminate variable and place in next column.

E.g., 0000 vs. 0100 yields 0-00
0000 vs. 1000 yields -000

When used in a combination, mark with a check. If cannot be combined, mark with a star. These are the prime implicants.

Repeat until no further combinations can be made.

Implication Table		
Column I	Column II	Column III
0000	0-00 * -000 *	01-- *
0100 1000	010- 01-0	-1-1 *
0101 0110 1001 1010	100- * 10-0 * 01-1 -101	
0111 1101	011- 1-01 *	
1111	-111 11-1	

CAD Tools for Simplification

Prime Implicant Chart

	4	5	6	8	9	10	13
0,4 (0-00)	X						
0,8 (-000)				X			
8,9 (100-)				X	X		
8,10 (10-0)				X		X	
9,13 (1-01)					X		X
4,5,6,7 (01-)	X	X	X				
5,7,13,15 (-1-1)		X					X

Stage 2: find smallest set of prime implicants that cover the ON-set

- recall that essential prime implicants must be in all covers

- another tabular method– the prime implicant chart

**rows = prime implicants
 columns = ON-set elements
 place an "X" if ON-set element is covered by the prime implicant**

CAD Tools for Simplification

Prime Implicant Chart

	4	5	6	8	9	10	13
0,4 (0-00)	X						
0,8 (-000)				X			
8,9 (100-)				X	X		
8,10 (10-0)				X		X	
9,13 (1-01)					X		X
4,5,6,7 (01-)	X	X	X				
5,7,13,15 (-1-1)		X					X

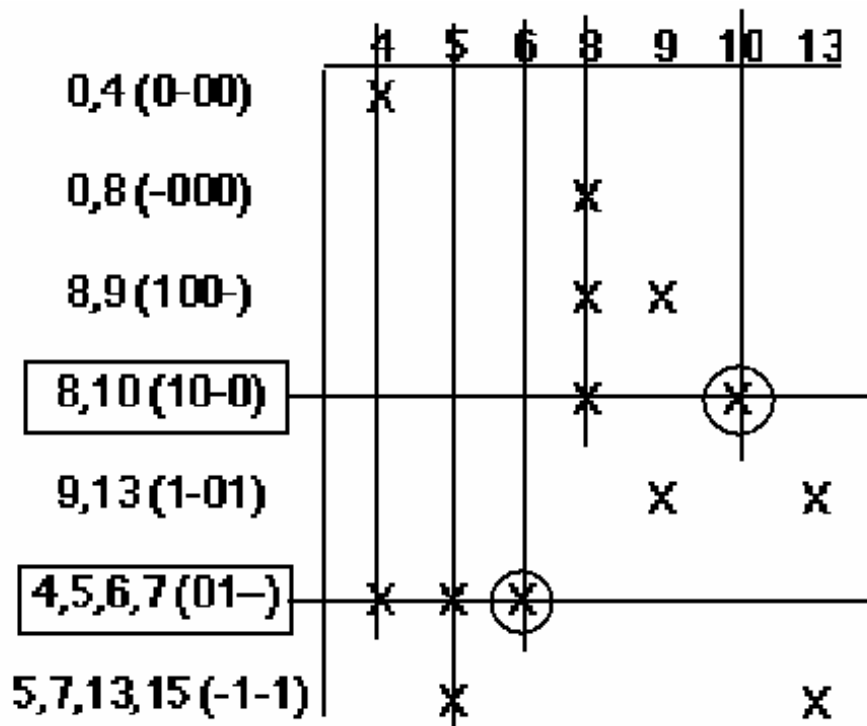
	4	5	6	8	9	10	13
0,4 (0-00)	X						
0,8 (-000)				X			
8,9 (100-)				X	X		
8,10 (10-0)				X		X	
9,13 (1-01)					X		X
4,5,6,7 (01-)	X	X	X				
5,7,13,15 (-1-1)		X					X

rows = prime implicants
columns = ON-set elements
place an "X" if ON-set element is covered by the prime implicant

If column has a single X, then the implicant associated with the row is essential. It must appear in minimum cover

CAD Tools for Simplification

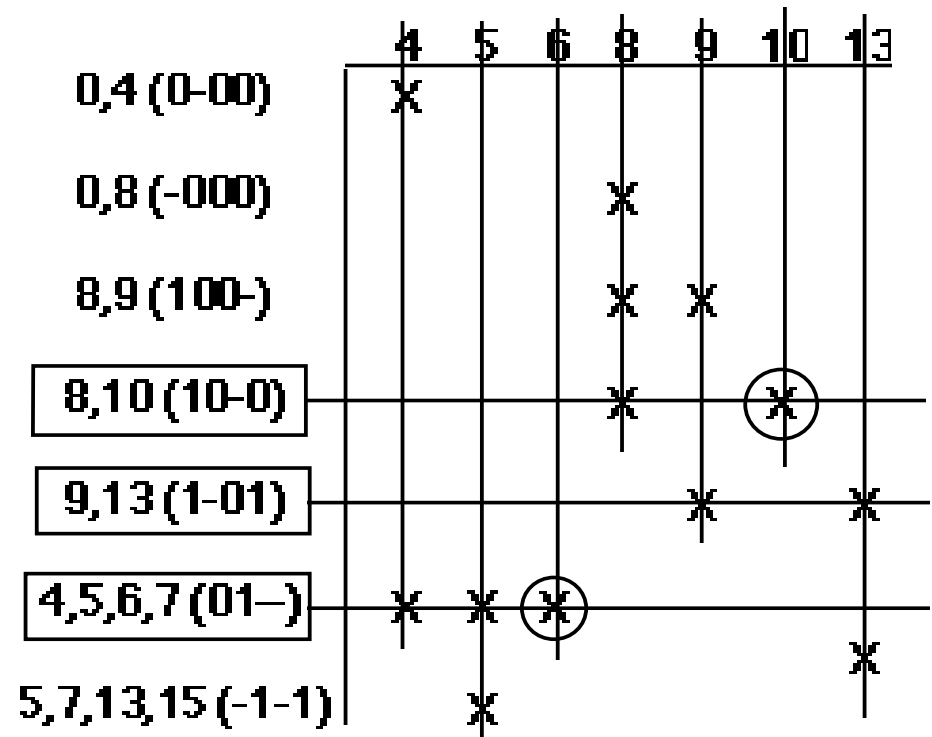
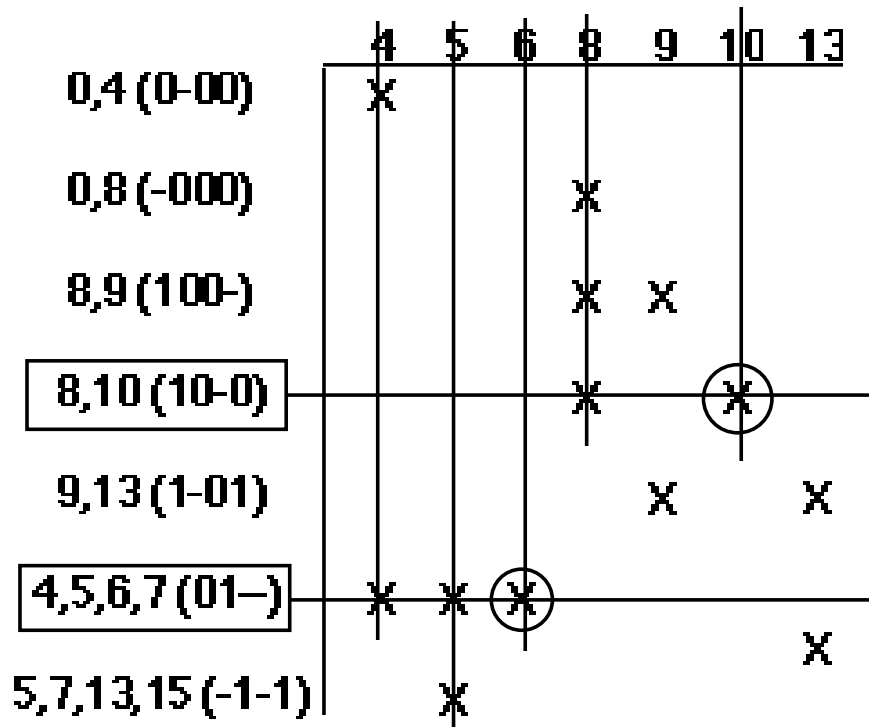
Prime Implicant Chart (Continued)



Eliminate all columns covered by essential primes

CAD Tools for Simplification

Prime Implicant Chart (Continued)



Eliminate all columns covered by essential primes

Find minimum set of rows that cover the remaining columns

$$f = A B' D' + A C' D + A' B$$

CAD Tools for Simplification

Quine-McCluskey Algorithm Continued

AB		A			
		00	01	11	10
C	CD	00	01	11	10
	00	X	1	0	1
	01	0	1	1	1
	11	0	X	X	0
	10	0	1	0	1
		B		D	

Prime Implicants:

$$0-00 = A' C' D'$$

$$-000 = B' C' D'$$

$$100- = A B' C'$$

$$10-0 = A B' D'$$

$$1-01 = A C' D$$

$$01-- = A' B$$

$$-1-1 = B D$$

Another view – in terms of a K-map

CAD Tools for Simplification

Quine-McCluskey Algorithm Continued

AB \ CD		A			
		00	01	11	10
C	00	X	1	0	1
	01	0	1	1	1
	11	0	X	X	0
	10	0	1	0	1
		B		D	

Prime Implicants:

$$0-00 = A' C' D'$$

$$-000 = B' C' D'$$

$$100- = A B' C'$$

$$10-0 = A B' D'$$

$$1-01 = A C' D$$

$$01-- = A' B$$

$$-1-1 = B D$$

The prime implicant chart identified the smallest set of prime implicants (rows in the chart) that covered the ON-set – just like we did visually with the K-map

Note: I always ask exactly one Quine-McCluskey problem on an exam each semester! You should understand the algorithm.

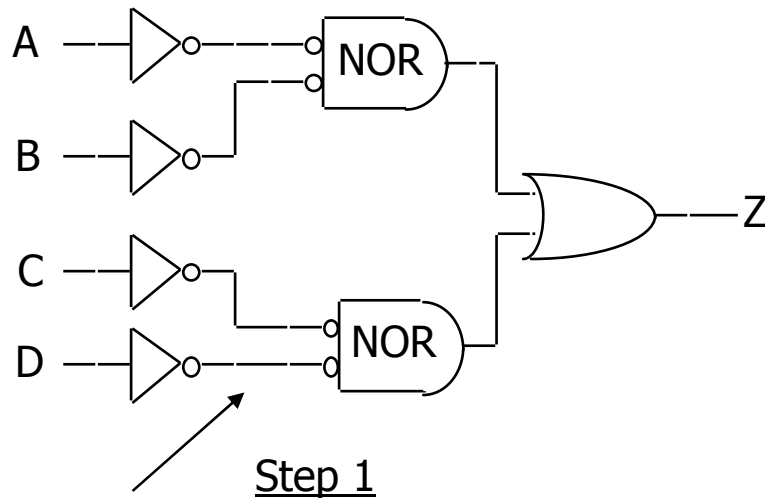
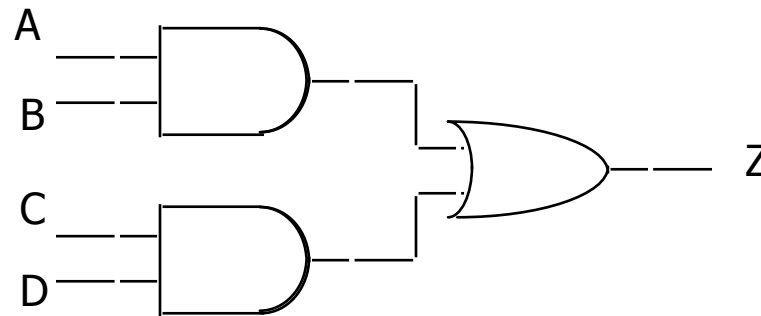
Real-World Logic Design

- Usually lots more than 6 inputs -- can't use Karnaugh maps or Quine McCluskey except in “toy” problems!
- Design correctness is more important than gate minimization
 - Use “higher-level language” to specify logic operations or use other computer aided techniques
- Use programs to manipulate logic expressions and minimize logic
- PALASM, ABEL, CUPL -- developed for PLDs
- VHDL, Verilog -- developed for ASICs

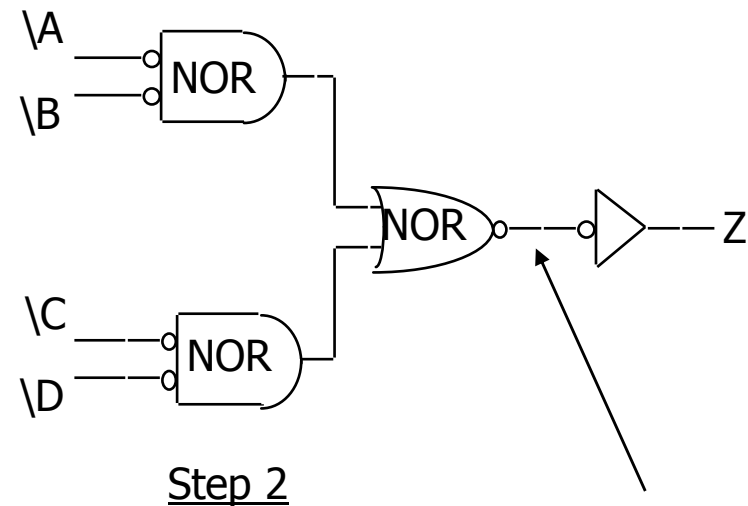
- We will use the Xilinx software to help us implement complex circuits on Programmable Logic Devices (PLDs) called Field Programmable Gate Arrays (FPGAs) without using a Hardware Description Language (HDL) – however, HDL is actually produced as an intermediate result by the Xilinx software

Conversion between forms

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



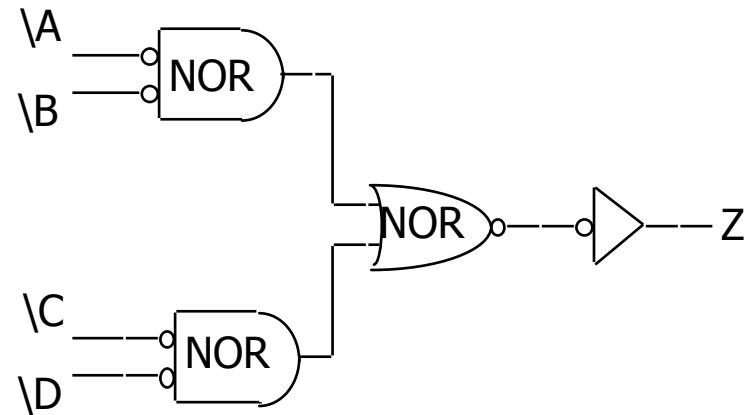
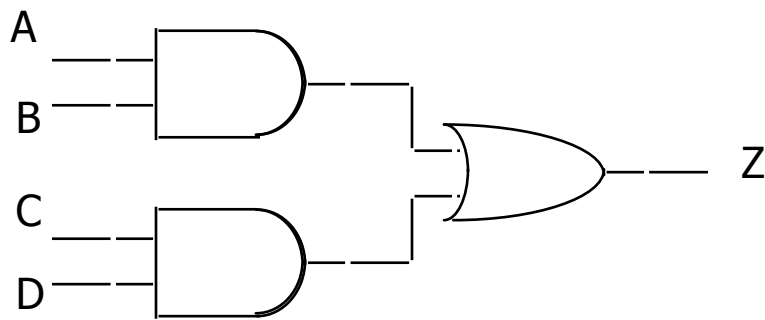
conserve
"bubbles"



conserve
"bubbles"

Conversion between forms (cont'd)

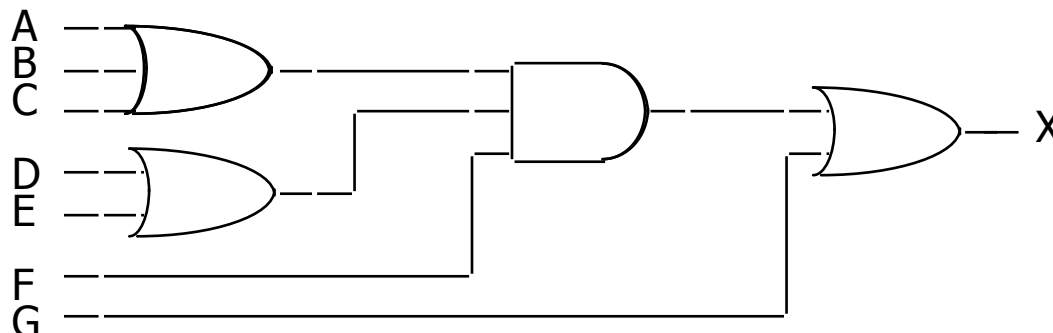
- Example: verify equivalence of two forms



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

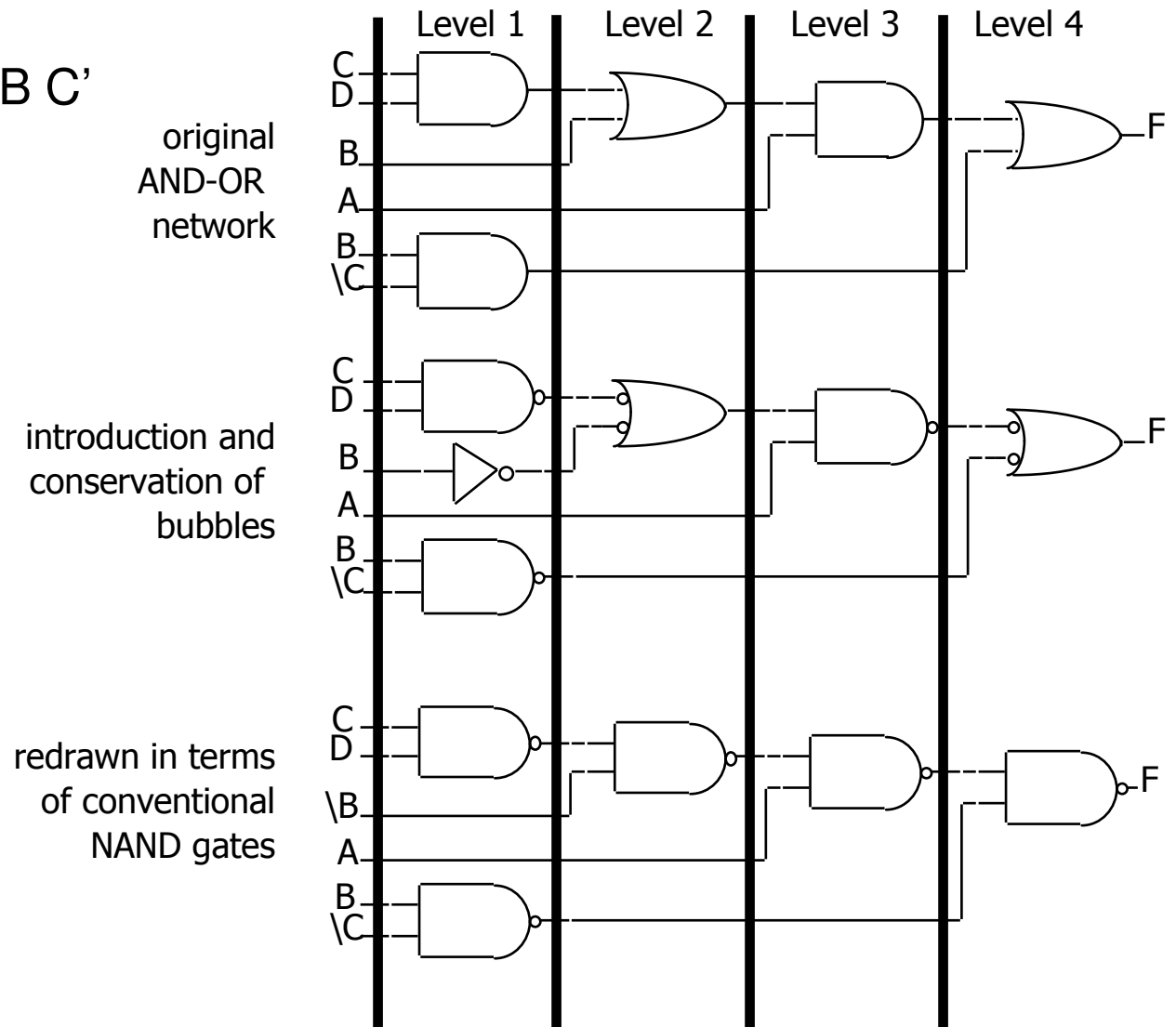
Multi-level logic

- $x = A D F + A E F + B D F + B E F + C D F + C E F + G$
 - ❑ reduced sum-of-products form – already simplified
 - ❑ 6 x 3-input AND gates + 1 x 7-input OR gate (that may not even 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)



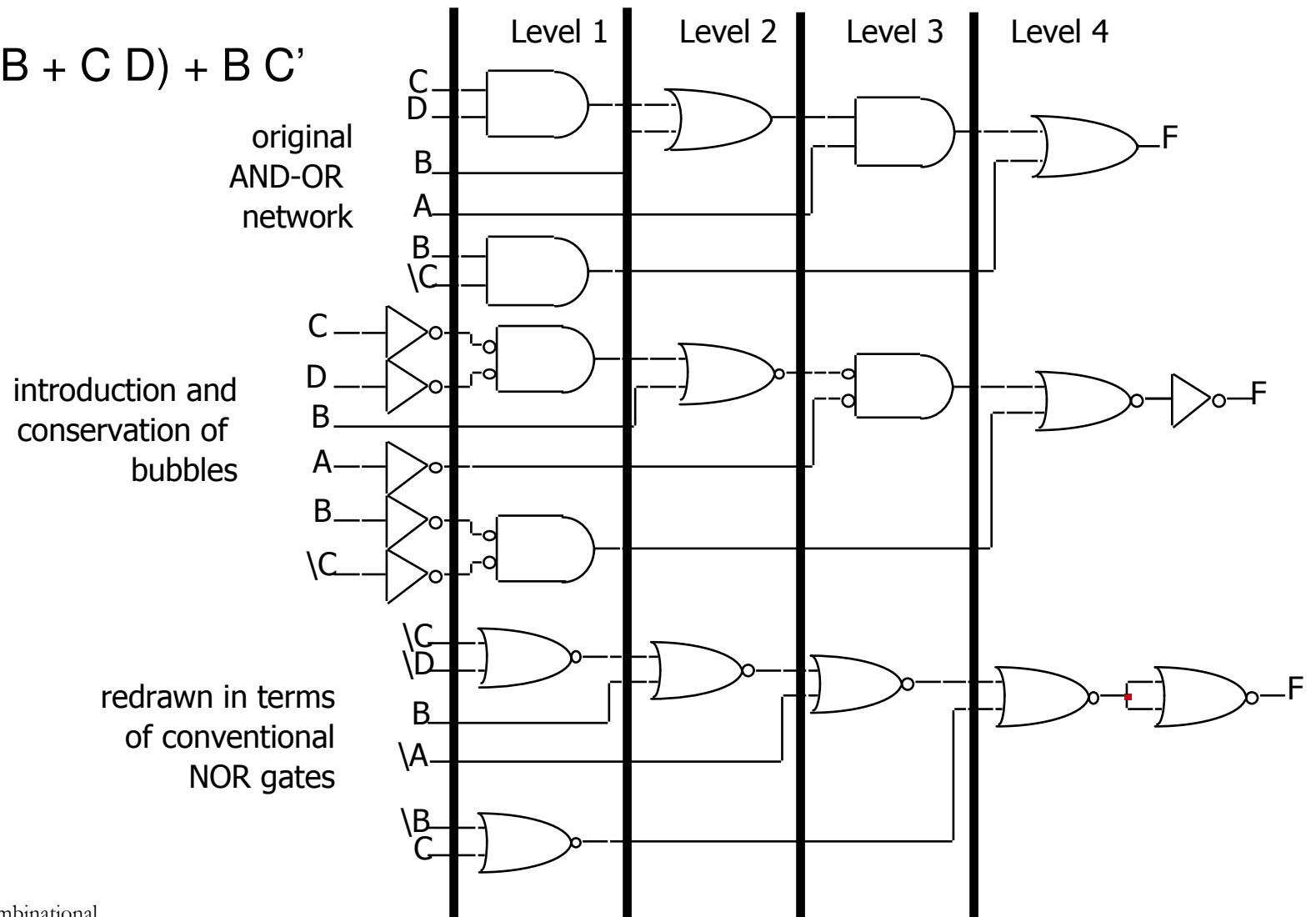
Conversion of multi-level logic to NAND gates

- $$F = A (B + C D) + B C'$$



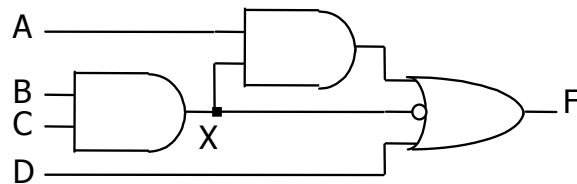
Conversion of multi-level logic to NORs

■ $F = A(B + CD) + BC'$

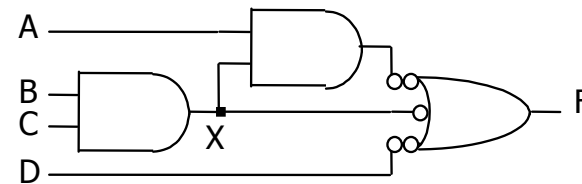


Conversion between forms

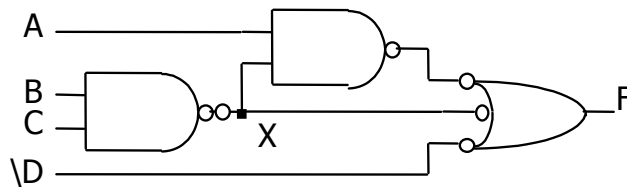
■ Example



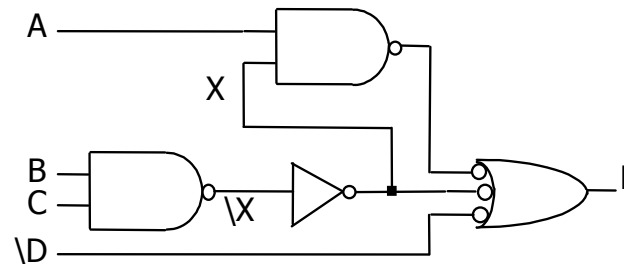
original circuit



add double bubbles to invert all inputs of OR gate



add double bubbles to invert output of AND gate



insert inverters to eliminate double bubbles on a wire

Summary for multi-level logic

■ Advantages

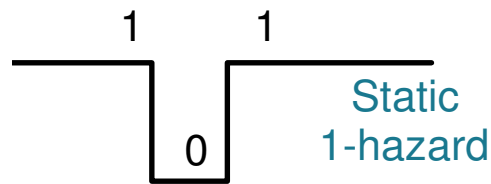
- ❑ circuits may be smaller
- ❑ gates have smaller fan-in
- ❑ circuits may be faster

■ Disadvantages

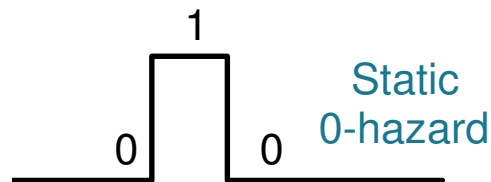
- ❑ more difficult to design
- ❑ tools for optimization are not as good as for two-level
- ❑ analysis is more complex

Time Response in Combinational Networks

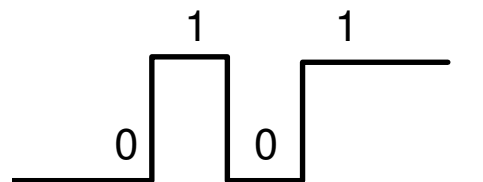
Hazards/Glitches and How to Avoid Them by Designing Hazard Free Circuits



Input change causes output to go from 1 to 0 to 1



Input change causes output to go from 0 to 1 to 0

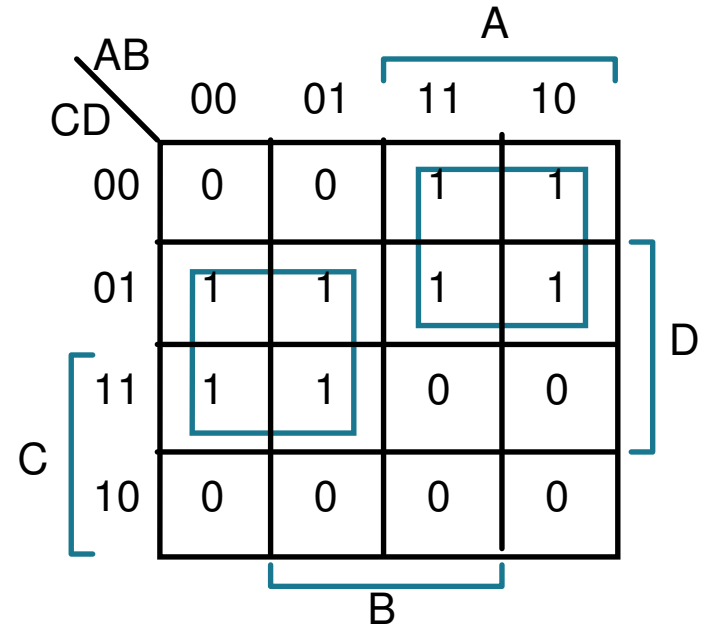
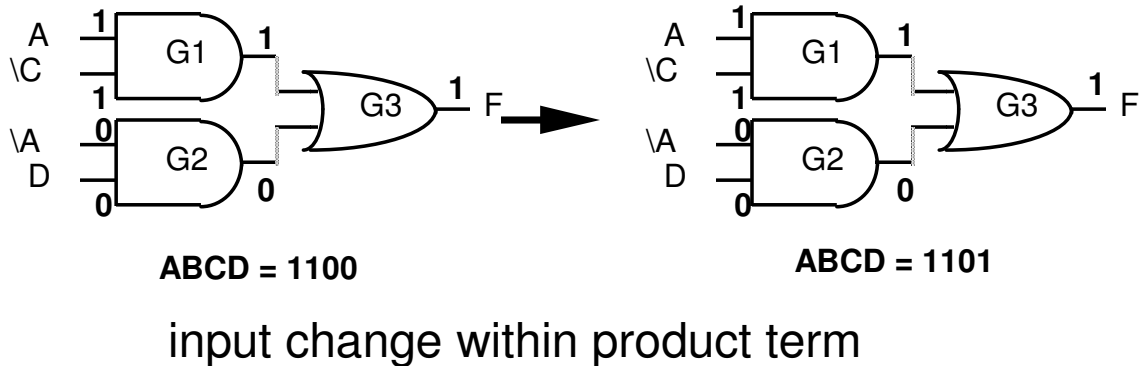


Input change causes a double change
from 0 to 1 to 0 to 1 OR
from 1 to 0 to 1 to 0

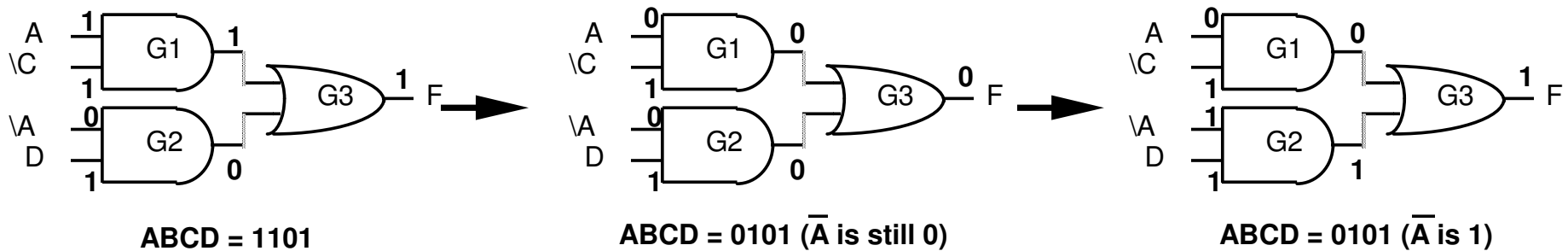
Kinds of Hazards

Time Response in Combinational Circuits

Glitch Example



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



input change that spans product terms
output changes from 1 to 0 to 1

Time Response in Combinational Networks

Glitch Example

General Strategy: add redundant terms

$$F = A' D + A C' \text{ becomes } A' D + A C' + C' D$$

Thus, the K-map can be used to detect static hazards in two-level SOP or POS logic circuits. The presence or absence of static hazards depends on the circuit design for the logic function.

A properly designed two-level SOP circuit has no static-0 hazards, but, it *may* have static-1 hazards.

A static-0 hazard would exist in such a circuit only if a variable and its complement were connected to the same AND gate – which would not make sense.

This eliminates the 1-hazard?

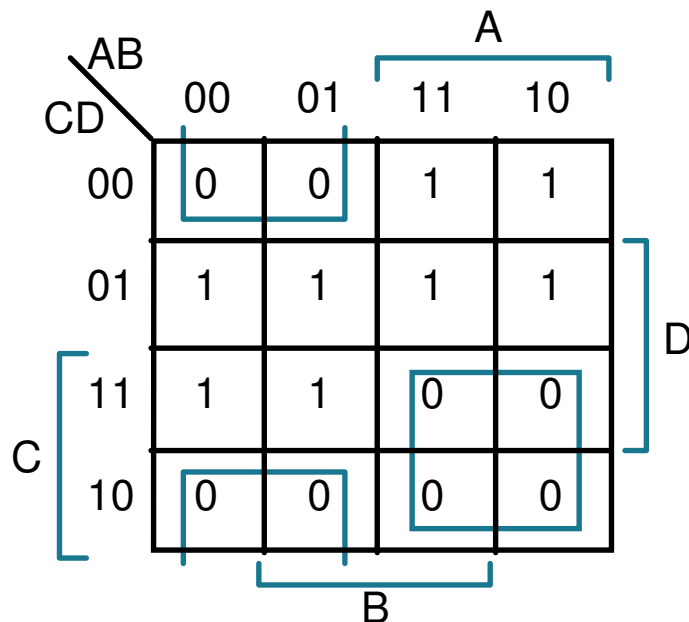
How about 0-hazards?

Time Response in Combinational Networks

Glitch Example

For POS circuits, eliminate static-0 hazards by working with the zeros in the K-map.

Cover transitions between prime implicants (groups of zeros) in a manner similar to that for SOP circuits.



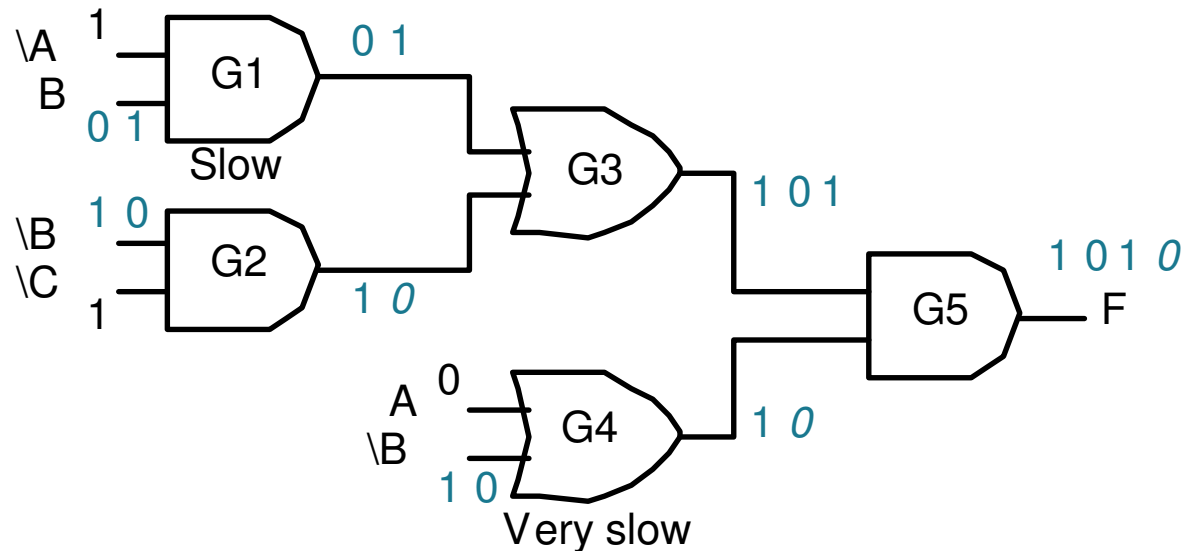
A properly designed two-level POS circuit has no static-1 hazards, but, it *may* have static-0 hazards.

A static-1 hazard would exist in such a circuit only if a variable and its complement were connected to the same OR gate – which would not make sense.

Time Response in Combinational Networks

Dynamic Hazards

Example with Dynamic Hazard



Three different paths from B or B' to output

$ABC = 000, F = 1$ to $ABC = 010, F = 0$

different delays along the paths:

G1 slow, G4 very slow

Handling dynamic hazards is very complex and beyond our scope

Hardware description languages

- Describe hardware at varying levels of abstraction
- Structural description
 - textual replacement for schematic
 - hierarchical composition of modules from primitives
- Behavioral/functional description
 - describe what module does, not how
 - synthesis generates circuit for module
- Simulation semantics
- HDLs are not emphasized in this course!
 - this information is here for completeness

HDLs

- Abel (circa 1983) - developed by Data-I/O
 - targeted to programmable logic devices
 - not good for much more than state machines
- ISP (circa 1977) - research project at CMU
 - simulation, but no synthesis
- Verilog (circa 1985) - developed by Gateway (absorbed by Cadence)
 - similar to Pascal and C
 - delays is only interaction with simulator
 - fairly efficient and easy to write
 - IEEE standard
- VHDL (circa 1987) - DoD sponsored standard
 - similar to Ada (emphasis on re-use and maintainability)
 - simulation semantics visible
 - very general but verbose
 - IEEE standard

Verilog

- Supports structural and behavioral descriptions
- Structural
 - explicit structure of the circuit
 - e.g., each logic gate instantiated and connected to others
- Behavioral
 - program describes input/output behavior of circuit
 - many structural implementations could have same behavior
 - e.g., different implementation of one Boolean function
- We'll mostly be using behavioral Verilog in Aldec ActiveHDL
 - rely on schematic when we want structural descriptions

Structural model

```
module xor_gate (out, a, b);  
    input      a, b;  
    output     out;  
    wire      abar, bbar, t1, t2;  
  
    inverter  invA (abar, a);  
    inverter  invB (bbar, b);  
    and_gate  and1 (t1, a, bbar);  
    and_gate  and2 (t2, b, abar);  
    or_gate   or1 (out, t1, t2);  
  
endmodule
```

Comparator example

```
module Compare1 (Equal, Alarger, Blarger, A, B);
    input      A, B;
    output     Equal, Alarger, Blarger;

    assign #5 Equal = (A & B) | (~A & ~B);
    assign #3 Alarger = (A & ~B);
    assign #3 Blarger = (~A & B);
endmodule
```

More complex behavioral model

```
module life (n0, n1, n2, n3, n4, n5, n6, n7, self, out);
  input      n0, n1, n2, n3, n4, n5, n6, n7, self;
  output     out;
  reg        out;
  reg [7:0] neighbors;
  reg [3:0] count;
  reg [3:0] i;

  assign neighbors = {n7, n6, n5, n4, n3, n2, n1, n0};

  always @(neighbors or self) begin
    count = 0;
    for (i = 0; i < 8; i = i+1) count = count + neighbors[i];
    out = (count == 3);
    out = out | ((self == 1) & (count == 2));
  end
endmodule
```

Hardware description languages vs. programming languages

- Program structure
 - instantiation of multiple components of the same type
 - specify interconnections between modules via schematic
 - hierarchy of modules (only leaves can be HDL in Aldec ActiveHDL)
- Assignment
 - continuous assignment (logic always computes)
 - propagation delay (computation takes time)
 - timing of signals is important (when does computation have its effect)
- Data structures
 - size explicitly spelled out - no dynamic structures
 - no pointers
- Parallelism
 - hardware is naturally parallel (must support multiple threads)
 - assignments can occur in parallel (not just sequentially)

Hardware description languages and combinational logic

- Modules - specification of inputs, outputs, bidirectional, and internal signals
- Continuous assignment - a gate's output is a function of its inputs at all times (doesn't need to wait to be "called")
- Propagation delay- concept of time and delay in input affecting gate output
- Composition - connecting modules together with wires
- Hierarchy - modules encapsulate functional blocks