
Ch 3. Working with Combinational Logic

Working with combinational logic

- Introduction

- Two-Level Simplification
- Automating Two-Level Simplification
- Multilevel Logic Networks
- Time Response in Combinational Networks
- Hardware Description Languages

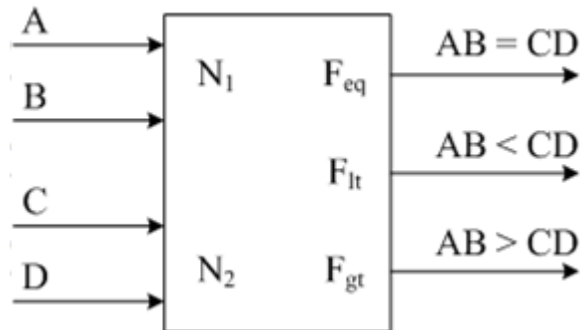
Two-Level Simplification

- Design Examples
 - Two-bit comparator
 - Two-bit binary adder
 - BCD increment-by-1 function
- Formalizing the Process of Boolean Minimization
 - Algorithm for two-level simplification
 - Application of the step-by-step algorithm
- K-Maps Revisited : Five- and Six-Variable Functions
 - Five-variable K-maps

Design Examples – Two-bit comparator

- The behavior of two-bit comparator

<Block diagram>



We need 4-variable K-maps
for each of 3 output functions

<Truth table>

A	B	C	D	F_{eq}	F_{lt}	F_{gt}
0	0	0	0	1	0	0
		0	1	0	1	0
		1	0	0	1	0
		1	1	0	1	0
0	1	0	0	0	0	1
		0	1	1	0	0
		1	0	0	1	0
		1	1	0	1	0
1	0	0	0	0	0	1
		0	1	0	0	1
		1	0	1	0	0
		1	1	0	1	0
1	1	0	0	0	0	1
		0	1	0	0	1
		1	0	0	0	1
		1	1	1	0	0

Design Examples – Two-bit comparator (cont'd)

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

<K-map for F_{eq} >

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

<K-map for F_{lt} >

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

<K-map for F_{gt} >

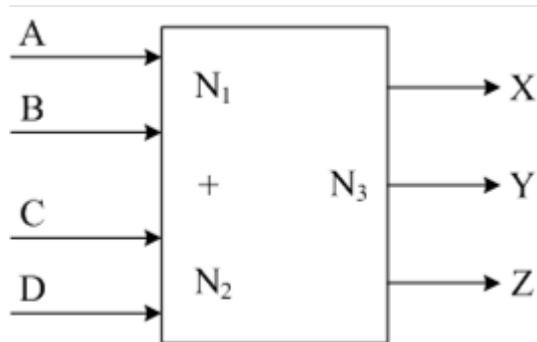
$$\begin{aligned}
 F_{eq} &= A'B'C'D' + A'BC'D + AB'CD' + ABCD \\
 &= A'C' (B'D' + BD) + AC (B'D' + BD) \\
 &= (A'C' + AC) (B'D' + BD) \\
 &= (A \oplus C)' (B \oplus D)' \\
 &= (A \equiv C) (B \equiv D)
 \end{aligned}$$

$$\begin{aligned}
 F_{lt} &= A'B'D + B'CD + A'C \\
 F_{gt} &= AC' + ABD' + BC'D'
 \end{aligned}$$

Design Examples – Two-bit binary adder

- The behavior of two-bit binary adder

<Block diagram>



X represents
the most significant bit.

<Truth table>

A	B	C	D	X	Y	Z
0	0	0	0	0	0	0
		0	1	0	0	1
		1	0	0	1	0
		1	1	0	1	1
0	1	0	0	0	0	1
		0	1	0	1	0
		1	0	0	1	1
		1	1	1	0	0
1	0	0	0	0	1	0
		0	1	0	1	1
		1	0	1	0	0
		1	1	1	0	1
1	1	0	0	0	1	1
		0	1	1	0	0
		1	0	1	0	1
		1	1	1	1	0

Design Examples – Two-bit binary adder (cont'd)

	A			
AB \ CD	00	01	11	10
00	0	0	0	0
01	0	0	1	0
11	0	1	1	1
10	0	0	1	1
	B			

Groupings for X:
 - A 4-element group (two literals) covering cells (11,00), (11,01), (11,11), (11,10)
 - Two 2-element groups (three literals) covering cells (01,11), (10,11) and (11,01), (10,01)

two 2-element groups (three literals),
one 4-element group (two literals) :

$$X = AC + BCD + ABD$$

	A			
AB \ CD	00	01	11	10
00	0	1	1	0
01	1	0	0	1
11	1	0	0	1
10	0	1	1	0
	B			

Groupings for Z:
 - Two 4-element groups (two literals) covering cells (01,00), (01,01), (01,11), (01,10) and (00,01), (00,11), (10,01), (10,11)

two 4-element groups (two literals) :

$$Z = BD' + B'D = B \oplus D$$

Design Examples – Two-bit binary adder (cont'd)

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

two 2-element groups (three literals) :
 $A'B'C$, $AB'C'$

four single-element groups (four literals) :
 $A'BC'D$, $A'BCD'$, $ABC'D'$, $ABCD$

↙ Factoring

$$A'B'C + AB'C' = B' (A'C + AC') = B' (A \oplus C)$$

$$A'BC'D + A'BCD' = A'B (C \oplus D) \quad ABC'D' + ABCD = AB (C \oplus D)'$$

$$A'B (C \oplus D) + AB (C \oplus D)' = B (A \oplus C \oplus D)$$

$$Y = B' (A \oplus C) + B (A \oplus C \oplus D)$$

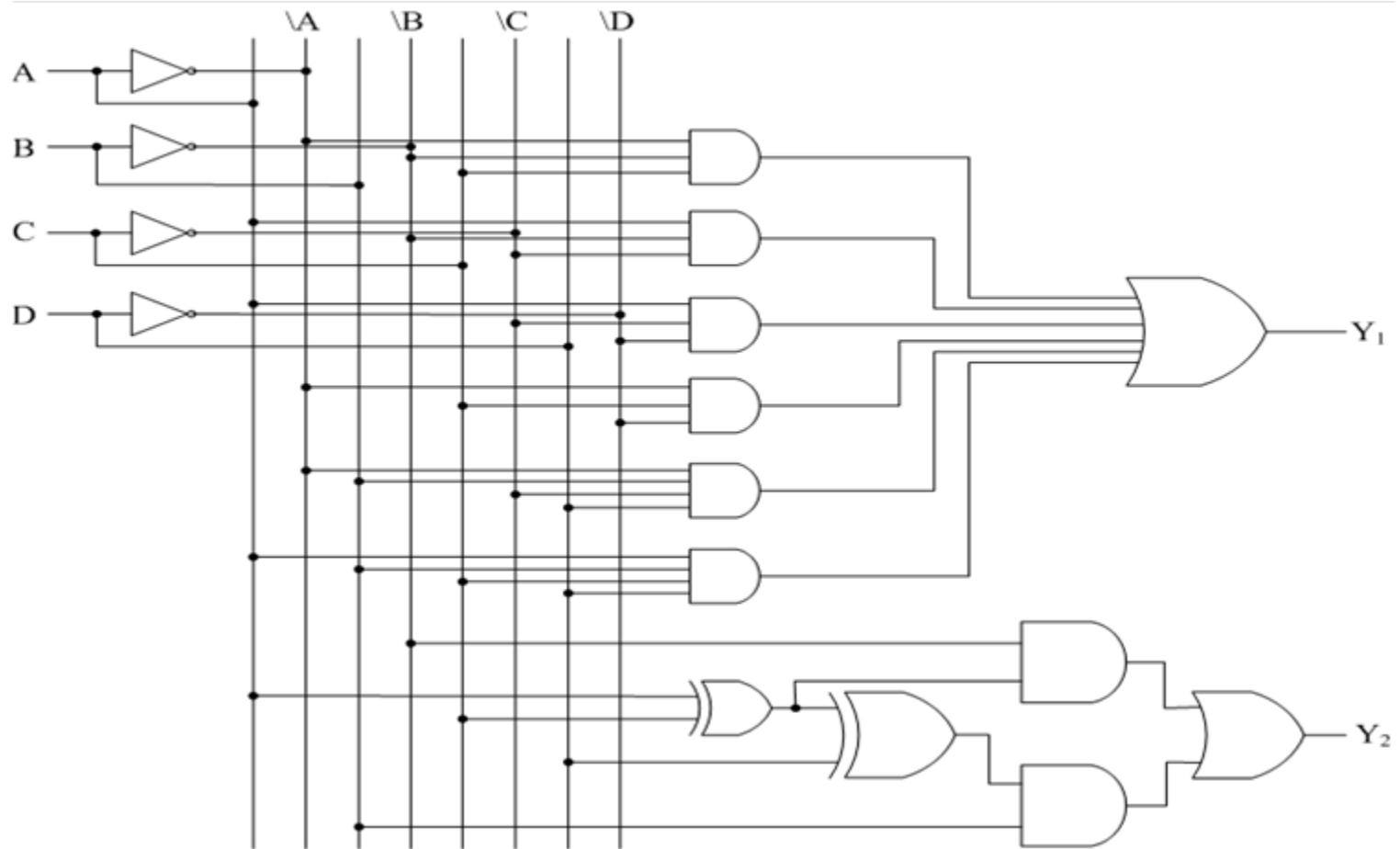
→ 5 gates, 7 literals

If only AND, OR, and NOT gates are allowed :

$$Y = A'B'C + AB'C' + AC'D' + A'CD' + A'BC'D + ABCD$$

→ 7 gates, 20 literals

Design Examples – Two-bit binary adder (cont'd)

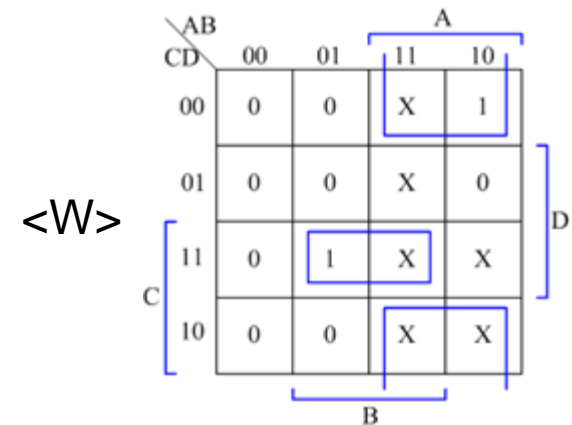


<Two alternative implementations of Y_2 >

Design Examples – BCD increment-by-1 function

- Truth table → Fig. 2.32

A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X



$$W = BCD + AD'$$

Design Examples – BCD increment-by-1 function

<W>

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

$$W = BCD + AD'$$

$$X = BD' + BC' + B'CD$$

$$Y = A'C'D + CD'$$

$$Z = D'$$

<Y>

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

<X>

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

<Z>

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

Formalizing the Process of Boolean Minimization

■ Definition of Terms

□ Implicant

- A single element of the on-set or any group of elements that can be combined together in a K-map

□ Prime implicant

- An implicant that cannot be combined with another one to eliminate a literal

□ Essential prime implicant

- A prime implicant that alone covers an element of on-sets
- must be part of the minimized expression as they are needed for any and all covers

■ Objective

- Grow implicant into prime implicants (minimize literals per term)
- Cover the ON-set with as few prime implicants as possible (minimize number of product terms)

Illustrating the Definitions

	A			
AB	00	01	11	10
CD	00	01	11	10
00	0	1	1	0
01	1	1	1	0
11	1	0	1	1
10	0	0	1	1

Prime Implicant : $A'B'D$, BC' , AC , $A'C'D$, AB , $B'CD$
 minimum cover : $A'B'D$, BC' , AC

Essential

	A			
AB	00	01	11	10
CD	00	01	11	10
00	0	0	1	0
01	1	1	1	0
11	0	1	1	1
10	0	0	1	1

Prime Implicant : BD , ABC' , ACD , $A'BC$, $A'C'D$
 minimum cover : ABC' , ACD , $A'BC$, $A'C'D$

Essential

	A			
AB	00	01	11	10
CD	00	01	11	10
00	0	0	0	0
01	0	1	1	0
11	1	1	1	1
10	1	0	1	1

Prime Implicant : BD , CD , AC , $B'C$
 minimum cover : BD , AC , $B'C$

Redundant

Two-level Simplification Algorithm

- A procedure for finding a minimum sum-of products expression from a K-map
 - Step 1 : Choose an element from the on-set
 - Step 2 : Find all of the “Maximal” groups of 1s and Xs adjacent to that element (This forms prime implicants)
 - Repeat Steps 1, 2 until all prime implicants have been found
 - Step 3 : Visit an element of the on-set
 - To find essential prime implicants
 - Repeat Step 3 until all essential prime implicants have been found
 - Step 4 : If there remain 1s uncovered by essential primes,
 - Select a minimum number of prime implicants that cover them

Application of the Step-by-step Algorithm

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

Example K-map

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

First 2 primes around $A'BC'D'$

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

1 prime is added around $A'BC'D$

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

1 prime is added around $ABC'D$

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

3 primes are added around $AB'C'D'$

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

Minimum cover (3 primes): $A'B$, $AB'D'$, $AC'D$

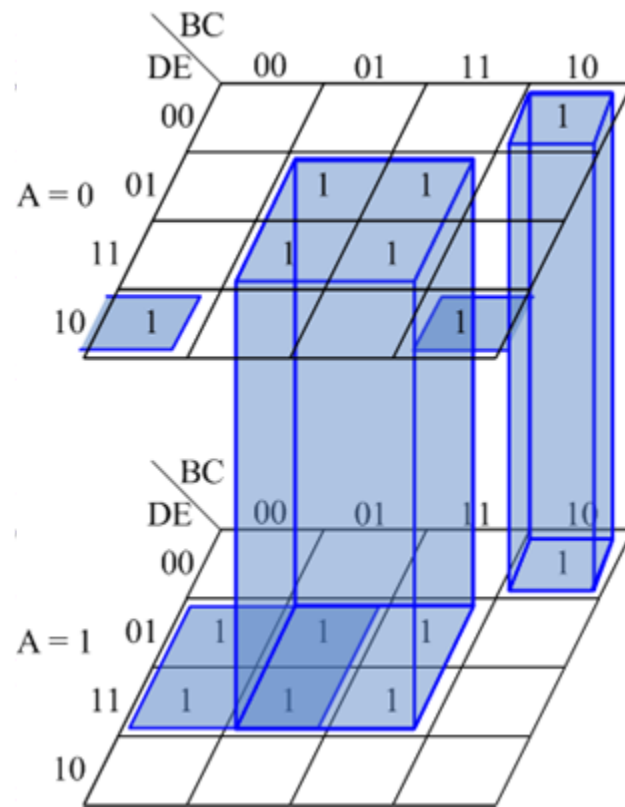
Essential

K-maps Revisited : Five-Variable Function

Example

$$F(A,B,C,D,E) = \sum m(2,5,7,8,10,13,15,17,19,21,23,24,29,31)$$

		BC			
		DE	00	01	11
A = 0	00	0	4	12	8
	01	1	5	13	9
	11	3	7	15	11
	10	2	6	14	10
A = 1	00	16	20	28	24
	01	17	21	29	25
	11	19	23	31	27
	10	18	22	30	26



Automating Two-Level Simplification

- Quine-McCluskey Method
 - Finding prime implicants
 - Finding the minimum cover

- *Espresso* Method
 - Algorithm used in *Espresso*
 - Example

- Realizing S-o-P and P-o-S Logic Networks
 - DeMorgan's law and pushing bubbles
 - Four conversion examples

Quine-McCluskey Method

Finding Prime Implicants

- Example Function : $F = \sum m(4,5,6,8,9,10,13) + d(0,7,15)$

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

✓ *implicant*
* *prime implicant*

- Stage 1 : Find all prime implicants
- Step 1 : Fill Column 1 with ON-set and DC-set minterm indices, grouped by the number of 1s
- Step 2 : Apply the Uniting theorem – Compare the elements in the 1st group against each element in the 2nd
 - e.g. 0000 vs. 0100 yields 0-00
 - 0000 vs. 1000 yields -000
 - When used in a combination, mark with a check (Implicant)
 - If cannot be combined, mark with a star (Prime implicant)
- Repeat until no further combinations can be made

Quine-McCluskey Method

Finding Prime Implicants (cont'd)

AB \ 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

Diagram illustrating the Karnaugh map for finding prime implicants. The map is a 4x4 grid with columns labeled AB (00, 01, 11, 10) and rows labeled CD (00, 01, 11, 10). The variables A, B, C, and D are indicated by brackets around the grid. Prime implicants are highlighted with blue boxes: a 2x2 square covering (00,01) and (01,11) in row 00; a 2x2 square covering (01,11) and (11,10) in row 00; a 2x2 square covering (01,11) and (11,10) in row 01; a 2x2 square covering (01,11) and (11,10) in row 11; a 2x2 square covering (01,11) and (11,10) in row 10; a 2x2 square covering (01,11) and (11,10) in row 10; a 2x2 square covering (01,11) and (11,10) in row 10; a 2x2 square covering (01,11) and (11,10) in row 10.

Prime implicants found by the Quine-McCluskey method :

$$0-00 = A'C'D' \quad -000 = B'C'D'$$

$$100- = AB'C' \quad 10-0 = AB'D'$$

$$1-01 = AC'D \quad 01-- = A'B$$

$$-1-1 = BD$$

Quine-McCluskey Method

Finding the Minimum Cover

- Stage 2 : Find the minimum cover – find the smallest collection of prime implicants that cover the complete on-set of the function through the **prime implicant chart**

(a) Initial 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

rows = prime implicants,
cols = ON-set elements

If an ON-set element is covered by the prime implicant, place a "X."

(b) Essential prime implicants

	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

If column has a single X, the implicant associated with the row is essential.

And it must be in the minimum cover.

Quine-McCluskey Method

Finding the Minimum Cover (cont'd)

(c) Covered minterms

	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

Eliminate all columns covered by essential primes.

Eliminate all rows covered by a set of essential primes.

(d) Final configuration

	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

Find minimum set of rows that cover the remaining columns.

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

Espresso Method

Algorithm Used in *Espresso*

- 1. **EXPAND** : Expand implicants to their maximum size
 - Implicants covered by an expanded implicant are removed from further consideration
 - Quality of results depends on the order and direction of implicant expansion
- 2. **IRREDUNDANT COVER**
 - An irredundant cover is extracted from the expanded implicants
- 3. **REDUCE** : Reduce prime implicants to the smallest size that still cover ON-set
- 4. Repeat sequence REDUCE/ EXPAND/ IRREDUNDANT COVER
 - Continue repeating these steps as long as generated cover improves on the last solution

Espresso Method

Example

(a) Initial prime implicant

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

Diagram (a) shows a 4x4 Karnaugh map with four prime implicants circled in blue: a 2x2 square in the top-left (C'D), a 2x2 square in the middle-left (C'D), a 2x2 square in the bottom-right (C'D), and a 2x2 square in the middle-right (C'D). Brackets labeled A, B, and C indicate the column, row, and column groupings respectively.

Initial set of primes after executing step 1 and 2 for the first time

4 primes, irredundant cover, but not the minimum cover

(b) Result of REDUCE step

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

Diagram (b) shows the same Karnaugh map as (a), but with the four prime implicants from (a) removed. The remaining prime implicants are circled in blue: a 2x2 square in the top-left (C'D), a 2x2 square in the middle-right (C'D), and a 2x2 square in the bottom-right (C'D). Brackets labeled A, B, and C indicate the column, row, and column groupings respectively.

The result of the REDUCE step :
C'D and CD' are reduced (therefore, they are no longer primes)

Espresso Method

Example (cont'd)

(c) Result of EXPAND step

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

Diagram (c) shows a 4x4 Karnaugh map with variables AB (columns) and CD (rows). The map contains 1s in the following cells: (00,00), (01,00), (01,01), (11,01), (10,01), (11,11), (10,11), (00,10), (01,10), (11,10), (10,10). Blue boxes highlight prime implicants: a 2x2 square covering (00,00), (01,00), (01,01), (00,01); a 2x2 square covering (01,01), (11,01), (10,01), (11,11); a 2x2 square covering (11,01), (10,01), (11,11), (10,11); a 2x2 square covering (11,11), (10,11), (11,10), (10,10); a 2x2 square covering (00,10), (01,10), (11,10), (10,10); and a 2x2 square covering (01,10), (11,10), (10,10), (00,10). Brackets labeled A, B, and C indicate the column, row, and row groups respectively.

The result of the second iteration of EXPAND

Espresso guarantees that it never generates the same cover twice

(d) Result of IRREDUNDANT COVER step

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

Diagram (d) shows the same 4x4 Karnaugh map as in (c). Blue boxes highlight the extracted irredundant cover: a 2x2 square covering (00,00), (01,00), (01,01), (00,01); a 2x2 square covering (01,01), (11,01), (10,01), (11,11); and a 2x2 square covering (11,11), (10,11), (11,10), (10,10). Brackets labeled A, B, and C indicate the column, row, and row groups respectively.

The extracted IRREDUNDANT COVER result

Only 3 prime implicants : an improvement on the original result

Realizing S-o-P and P-o-S Logic Networks

■ DeMorgan's Law and Pushing Bubbles

$$(AB)' = (A' + B')$$

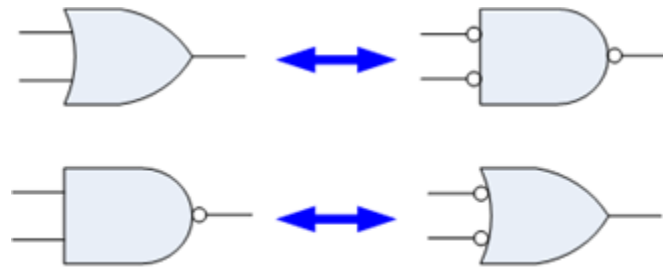
$$AB = (A' + B')$$

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

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

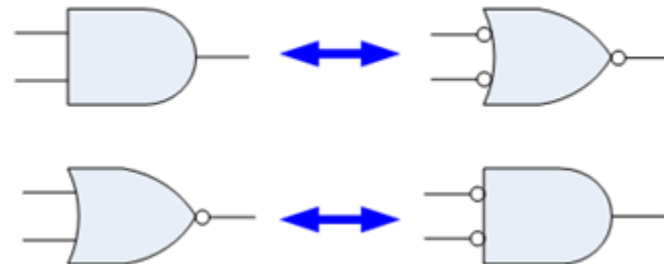
< OR/NAND equivalence >

A	\bar{A}	B	\bar{B}	$A+B$	$\overline{A \cdot B}$	$\bar{A} + \bar{B}$	$\overline{A \cdot B}$
0	1	0	1	0	0	1	1
0	1	1	0	1	1	1	1
1	0	0	1	1	1	1	1
1	0	1	0	1	1	0	0

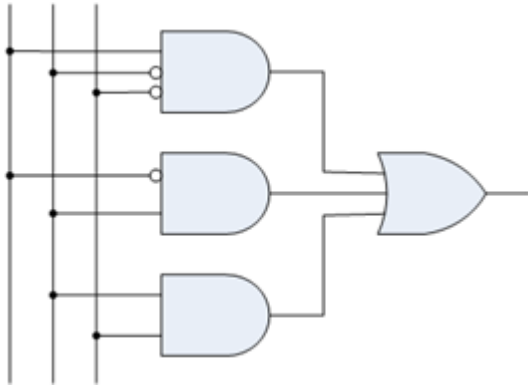


< AND/NOR equivalence >

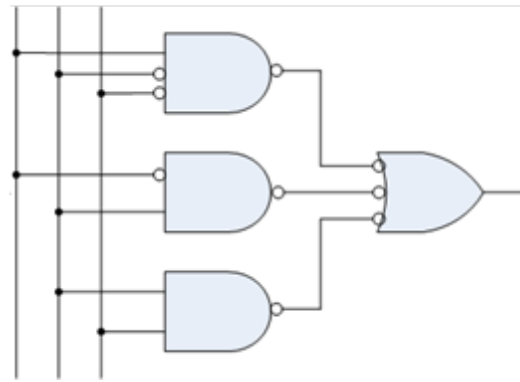
A	\bar{A}	B	\bar{B}	$A \cdot B$	$\overline{A + B}$	$\bar{A} \cdot \bar{B}$	$\overline{A + B}$
0	1	0	1	0	0	1	1
0	1	1	0	0	0	0	0
1	0	0	1	0	0	0	0
1	0	1	0	1	1	0	0



AND/OR Conversion to NAND/NAND Networks



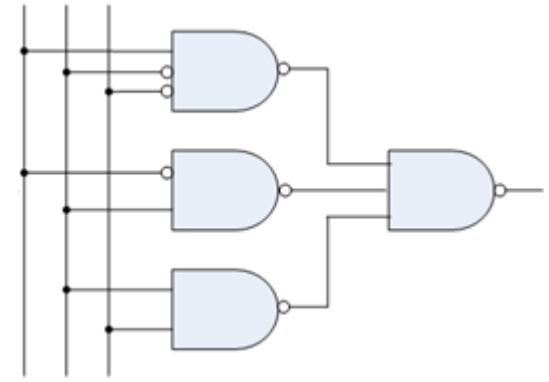
Initial AND/OR network



Conversion at the 1st level

1st-level AND gates are converted to their NAND equivalents

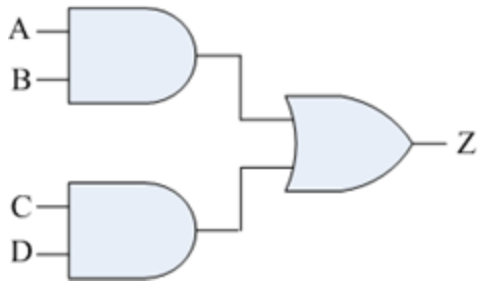
And complement the inputs to OR gates to conserve the circuits logic function



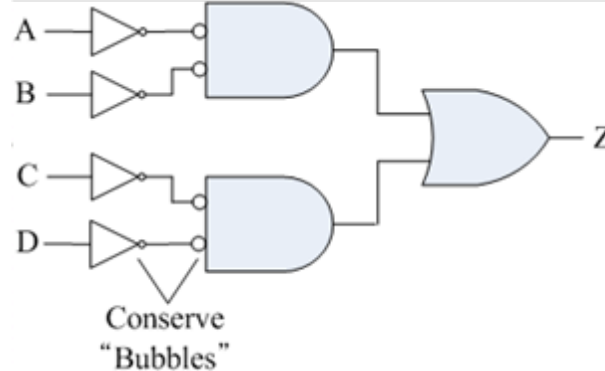
Conversion at the 2nd level

2nd level OR gate with complemented inputs is replaced by NAND gate

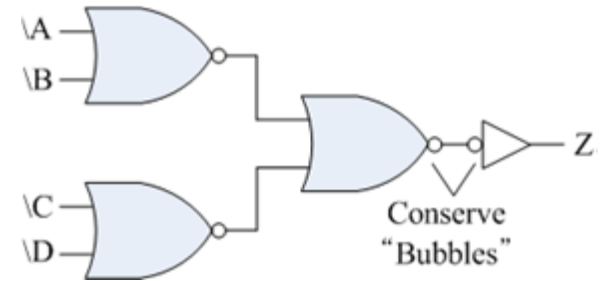
AND/OR Conversion to NOR/NOR Networks



Initial AND/OR network



Complemented inputs are created at the two AND gates



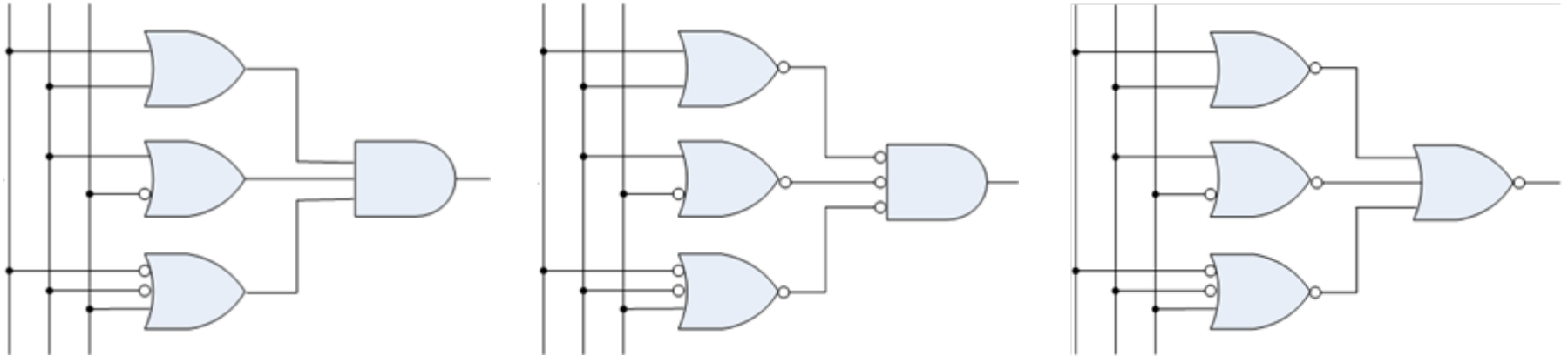
Two AND gates with complemented inputs are replaced by NOR gates

2nd level OR gate is converted to NOR gate after introducing a matching inverter

“Conserving Bubbles” :

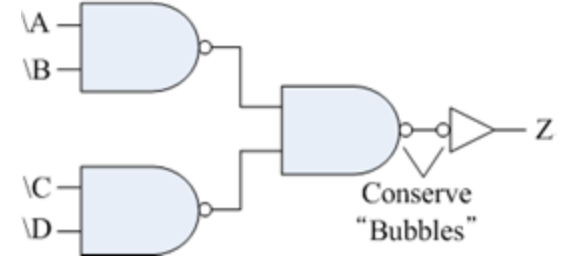
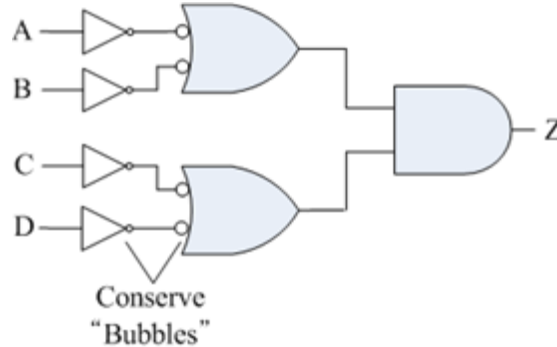
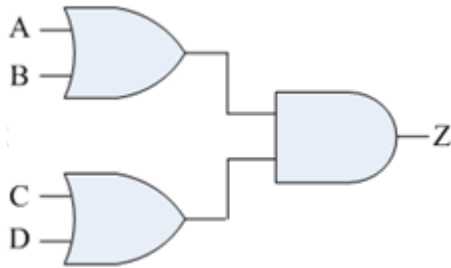
When a new inversion is introduced, it must be balanced by a complementary inversion

OR/AND Conversion to NOR/NOR Networks



Similar with "AND/OR Conversion to NAND/NAND Networks"

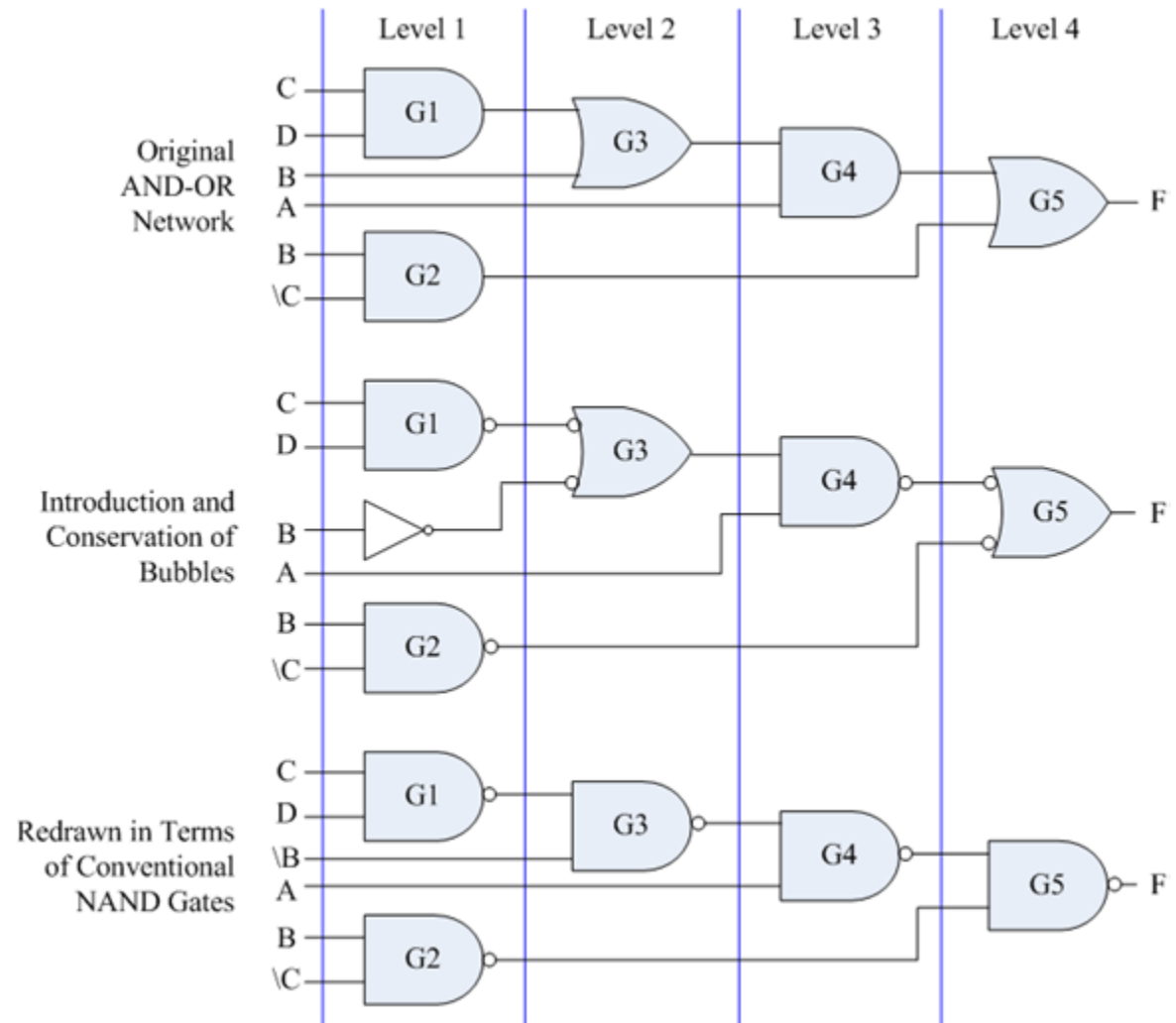
OR/AND Conversion to NAND/NAND Networks



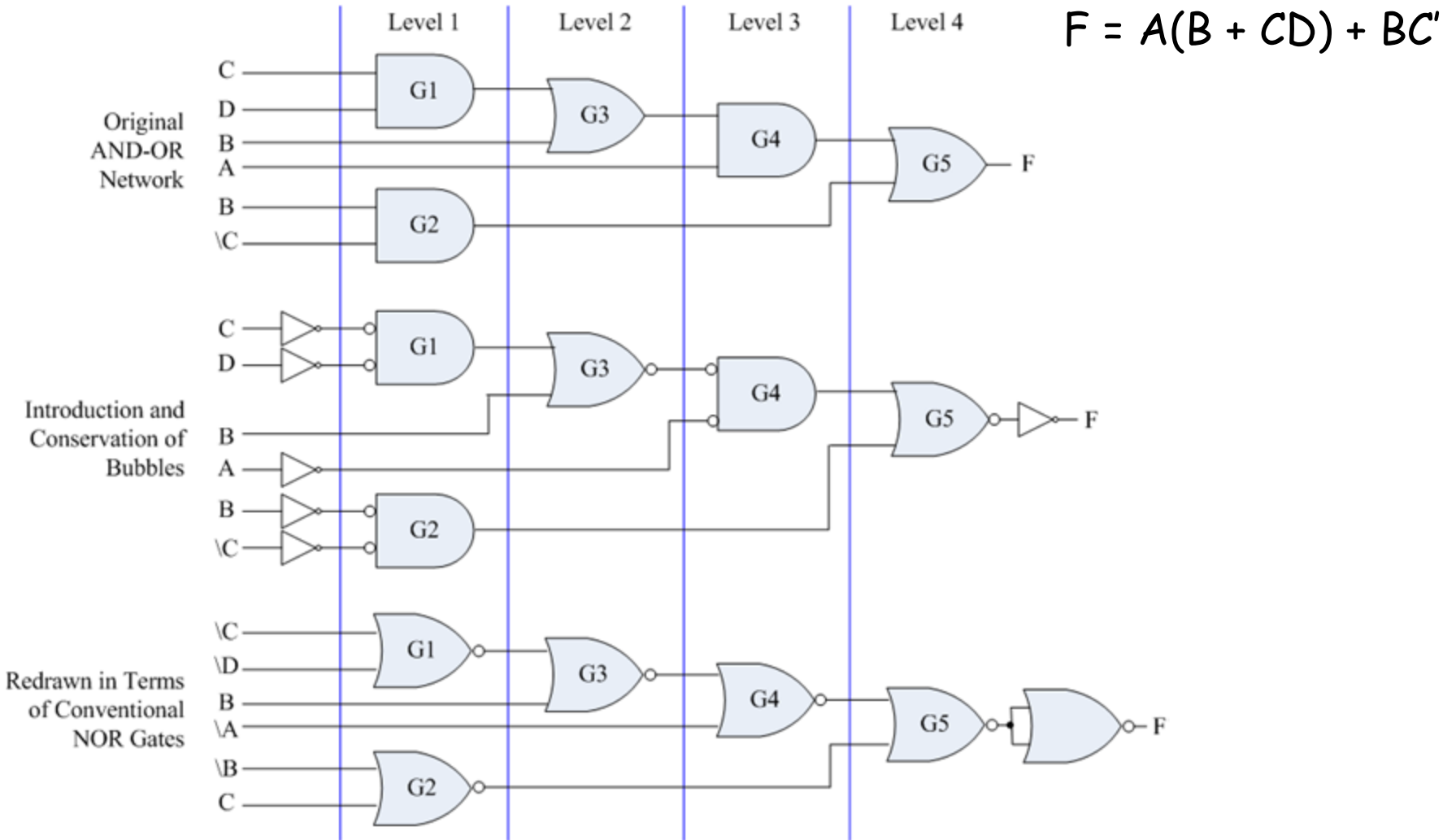
Similar with "AND/OR Conversion to NOR/NOR Networks"

Multilevel Logic Networks – Multilevel Conversion to NAND Gates

$$F = A(B + CD) + BC'$$



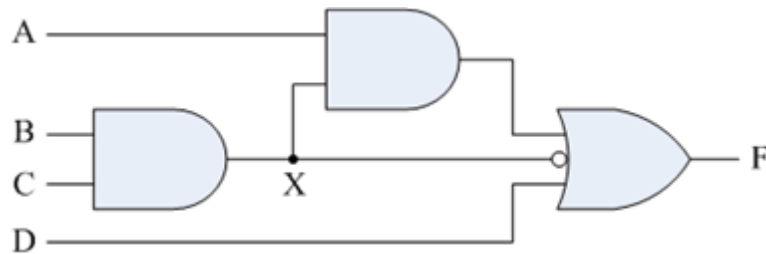
Multilevel Conversion to NOR Gates



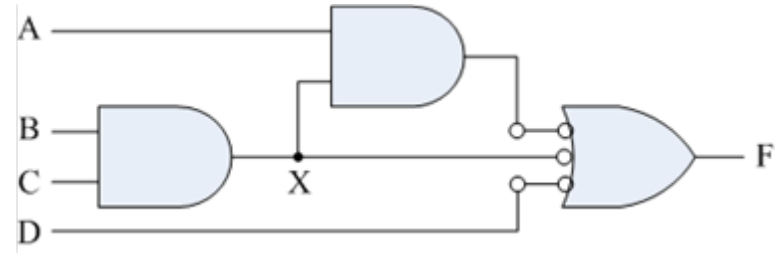
Non-Alternating NAND/NOR Multilevel Networks

$$F = AX + X' + D$$

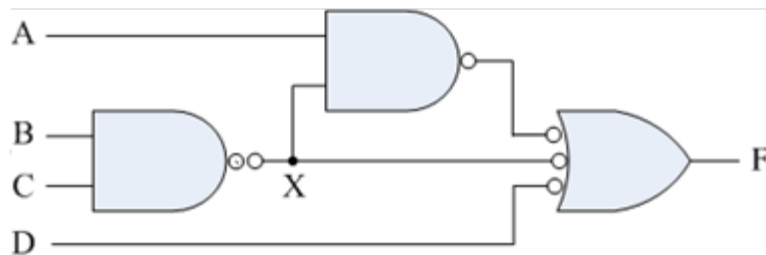
$$X = BC$$



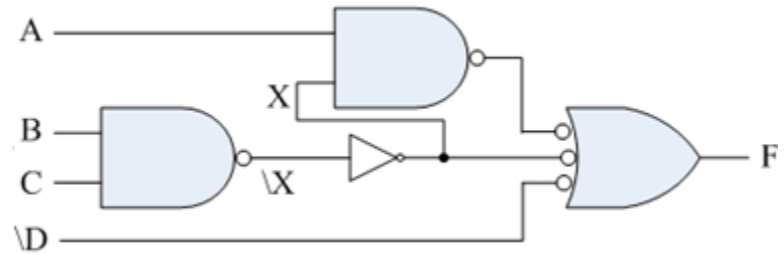
(a) Original Circuits



(b) Add Double Bubbles to Invert All Inputs of OR Gate



(c) Add Double Bubbles to Invert Output of AND Gate

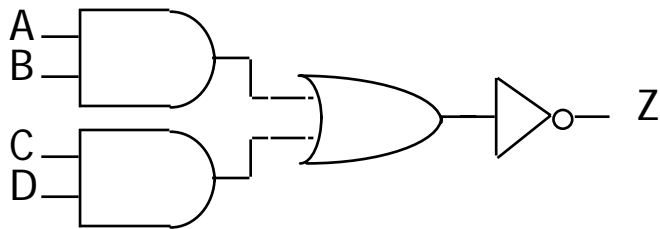


(d) Insert Inverters to Eliminate Double Bubbles on a Wire

AND-OR-Invert Gates

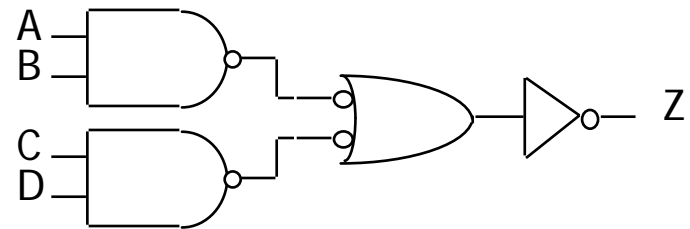
- AOI function: three stages of logic — AND, OR, Invert
 - multiple gates "packaged" as a single circuit block

logical concept



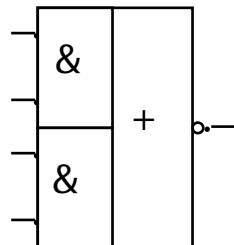
AND OR Invert

possible implementation

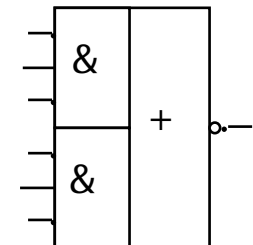


NAND NAND Invert

2x2 AOI gate symbol

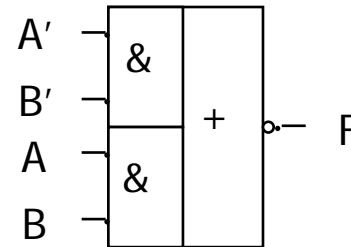


3x2 AOI gate symbol



Conversion to AOI Forms

- General procedure to place in AOI form
 - compute the complement of the function in sum-of-products form by grouping the 0s in the Karnaugh map
- Example: XOR implementation
 - $A \text{ xor } B = A' B + A B'$
 - AOI form:
 - $F = (A' B' + A B)'$



Examples of Using AOI Gates

■ Example:

- $F = A B + A C' + B C'$
- $F = (A' B' + A' C + B' C)'$
- Implemented by 2-input 3-stack AOI gate

- $F = (A + B) (A + C') (B + C')$
- $F = [(A' + B') (A' + C) (B' + C)]'$
- Implemented by 2-input 3-stack OAI gate

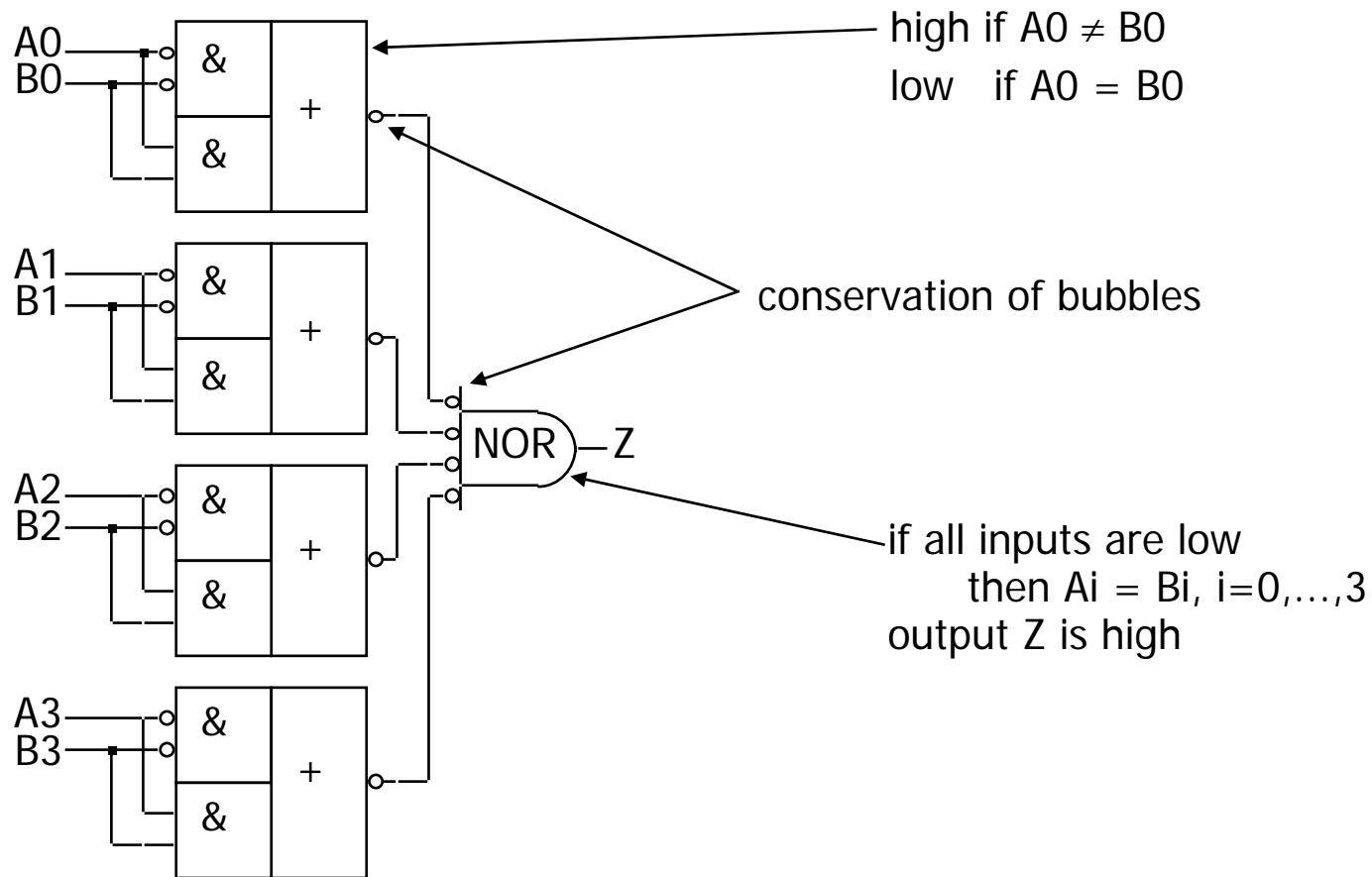
■ Example: 4-bit equality function

- $Z = (A_0 B_0 + A_0' B_0')(A_1 B_1 + A_1' B_1')(A_2 B_2 + A_2' B_2')(A_3 B_3 + A_3' B_3')$

each implemented in a single 2x2 AOI gate

Examples of Using AOI Gates (cont'd)

- Example: AOI implementation of 4-bit equality function



Time Response in Combinational Networks

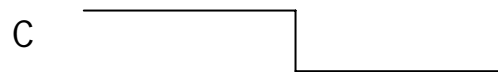
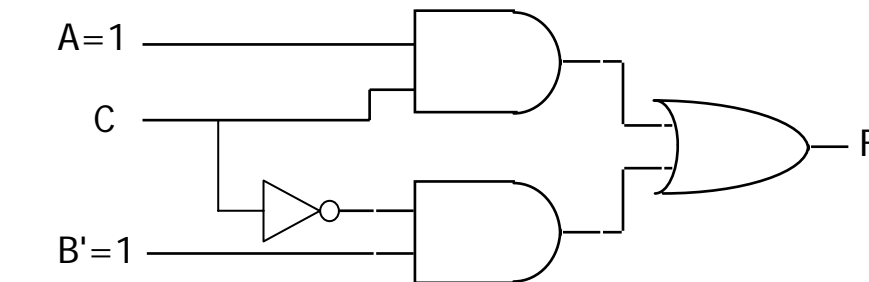
- Time response tells us about a circuit's dynamic behavior
 - Transient output changes: glitches
 - Logical error caused by glitches: a hazard
- It is important to visualize the behavior of a circuit as a function of time
 - Simulation tools can offer the time-based behavior of circuits
- Gate Delays
 - Defined in terms of minimum (best case), typical (average), and maximum (worst case) times.
 - Worst-case delay should always be considered
 - There are trade-offs between delay and power

Time Response in Combinational Networks

■ Example

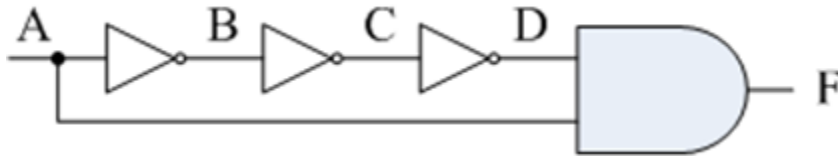
		A	
	1	0	1
C	0	1	1
		B	

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

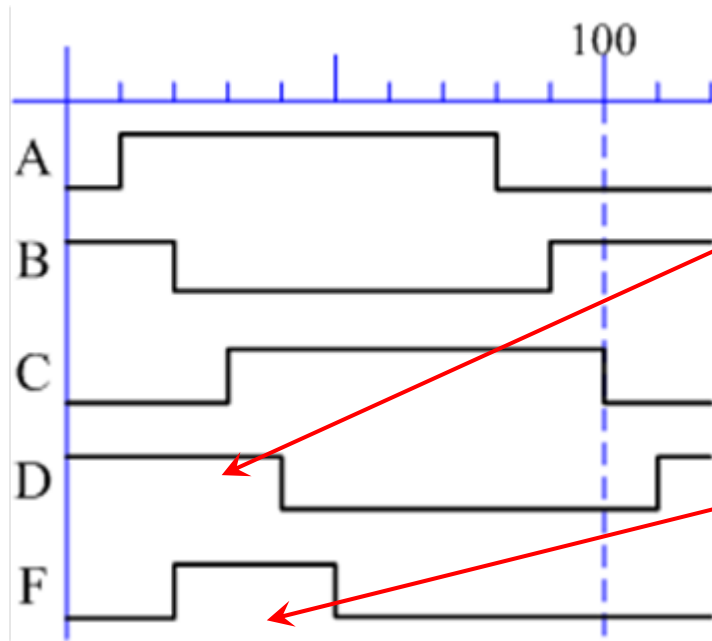


F=?

Timing Waveforms



- A pulse shaper
 - At a glance, $A A' = 0$
 - Delays matter

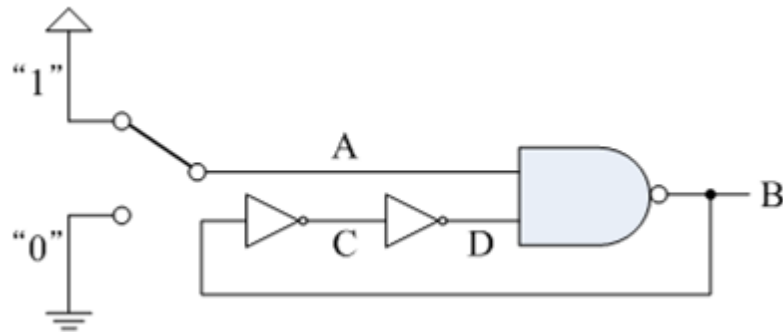


D remains high for three gate delays after A changes from low to high

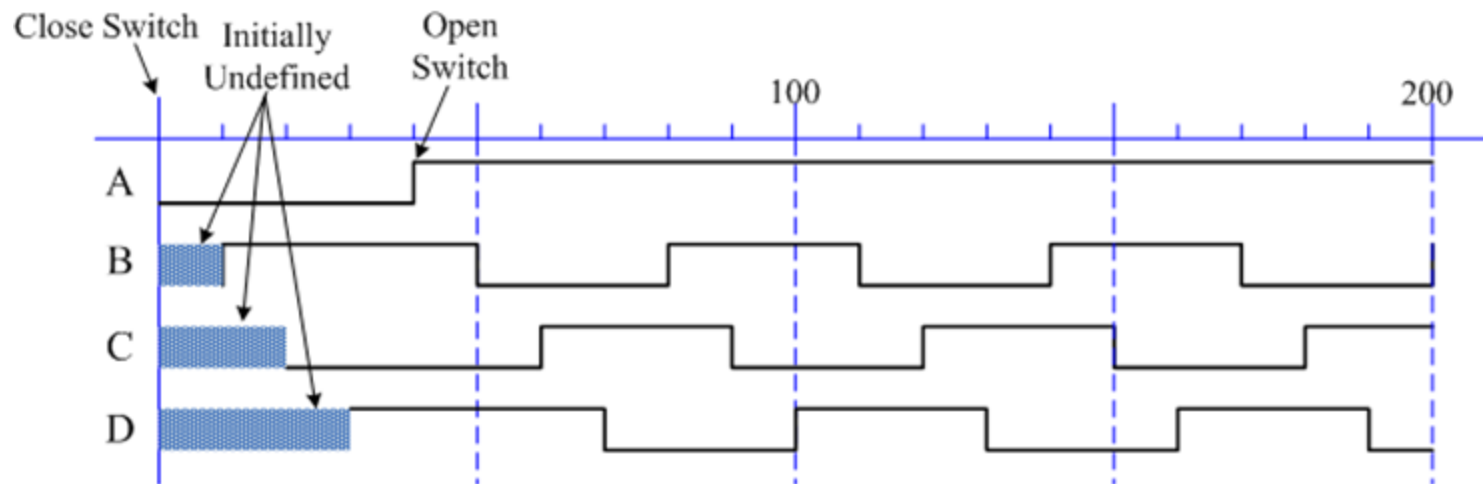
F is not always '0'

the pulse is exactly three inverter-delays wide

Analysis of a Pulse-Shaper Circuit



- Another Pulse-Shaper example



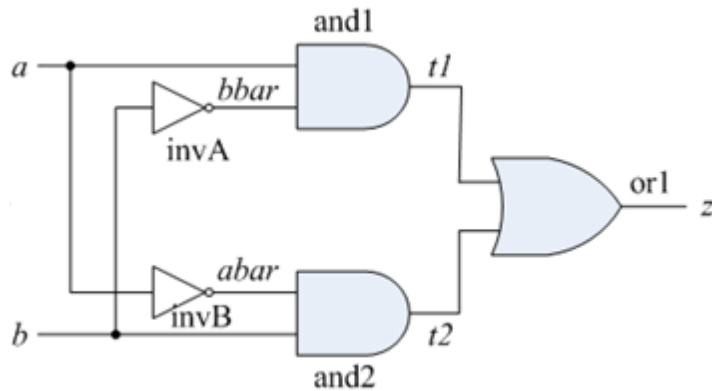
Hardware Description Languages

- Describe behavior
 - describe what module does, not how
 - synthesis generates circuit for module
- Describe structure
 - textual replacement for schematic
 - hierarchical composition of modules from primitives
- Describe timing
 - describe delay
- Describe concurrency
- Describe hardware at varying levels of abstraction
- Enable simulation
 - event-driven simulation

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 C
 - fairly efficient and easy to write
 - IEEE standard
- VHDL (circa 1987) - DoD sponsored standard
 - similar to Ada (emphasis on re-use and maintainability)
 - very general but verbose
 - IEEE standard

Describing Structure



XOR gate :

five gates and connecting wires

```
module xor_gate (a, b, z);  
  input      a, b;  
  output    z;  
  wire      abar, bbar, t1, t2;
```

Each gate is an *instance*
of another module

```
  inverter invA (abar, a);  
  inverter invB (bbar, b);  
  and_gate and1 (t1, a, bbar);  
  and_gate and2 (t2, b, abar);  
  or_gate  or1 (z, t1, t2);
```

```
endmodule
```

Describing Behavior

```
module xor_gate (a, b, z);  
    input      a, b;  
    output     z;  
    reg       z;  
  
    always @(a or b) begin  
        z = a ^ b;  
    end  
  
endmodule
```

always block ←

→ *sensitivity list*

- **always** block: specifies *when* and *how* the module behave
- *sensitivity list*: specifies when the block is executed (triggered by which signals)

Delay

```
module xor_gate (a, b, z);  
  input    a, b;  
  output   z;  
  
  assign #6 z = a ^ b;  
  
endmodule
```

*delay
statement*

```
module xor_gate (a, b, z);  
  input    a, b;  
  output   z;  
  reg      z;  
  
  always @(a or b) begin  
    #6 z = a ^ b;  
  end  
  
endmodule
```

endmodule

- The *delay* statement postpones the assignment of a new value to output
 - *delay* statements only make sense within a behavioral description

Event-Driven Simulation

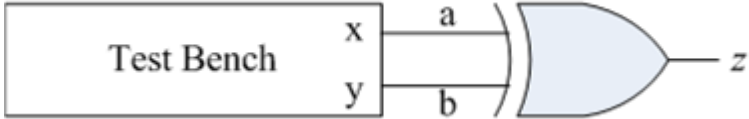
```
module test_bench (x, y);
    output x, y;
    reg x, y;

    initial begin
        x = 0; y = 0;
        #10;
        x = 0; y = 1;
        #10;
        x = 1; y = 0;
        #10;
        x = 1; y = 1;
        #10;
        $finish
    end
endmodule

module both_together (z);
    output z;
    wire w1, w2;

    test_bench tb1(w1, w2);
    xor_gate xor1(w1, w2, z);

    always @(z) begin
        $display("At time: %d with
        inputs:%b and %b, the output
        is: %b", $time, w1, w2, z);
    end
endmodule
```



< Schematic of an XOR gate connected to a stimulus generator >