Chapter 4. Simplification of Boolean Functions

Boolean Cubes and Boolean Functions

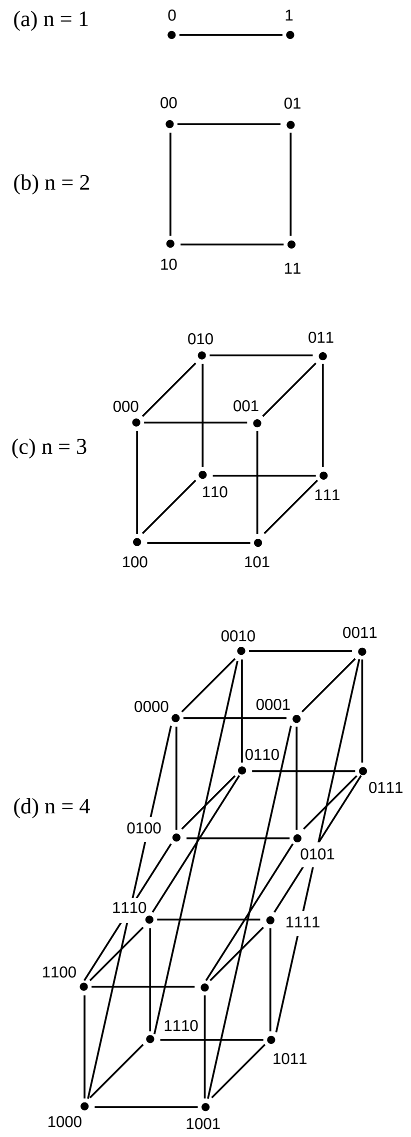


Figure 1: Boolean cubes [Gajski].

☞ A Boolean n-cube uniquely represents a Boolean function of n variables if each vertex is assigned a 1 (marked) or 0 (unmarked).

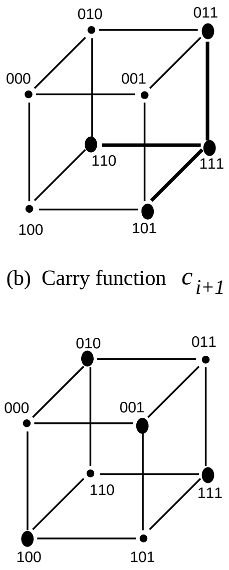
☞ Each vertex of the n-cube represents a minterm (a row in the truth table).

Example 1

Fig. 2 shows the truth table and the corresponding cube representations of the carry and sum functions. □

c_i	x_i	y_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a) Truth table



(b) Carry function c_{i+1}

(c) Sum function s_i

Figure 2: Boolean-cube representations for carry and sum functions [Gajski].

☞ (a) Each m-subcube of the n-cube represents 2^m minterms with the same $n - m$ literals, where $m < n$; (b) each m-subcube with 2^m 1-minterms represents a product term of $n - m$ literals.

☞ A prime implicant (PI) is a subcube (of 1-minterms) that is not contained in any other subcube (of 1-minterms); an essential prime implicant (EPI) is a PI that contains a 1-minterm that is not contained in any other PI.

Map Representation

- Complexity of digital circuit (gate count) $\propto$ complexity of algebraic expression (literal count).
- A function's truth-table representation is unique; its algebraic expression is not. Simplification by algebraic means is awkward (from algorithmic point of view).
- A Karnaugh map (K-map) is an array of squares each representing one minterm. Simplification by the map method is straightforward.

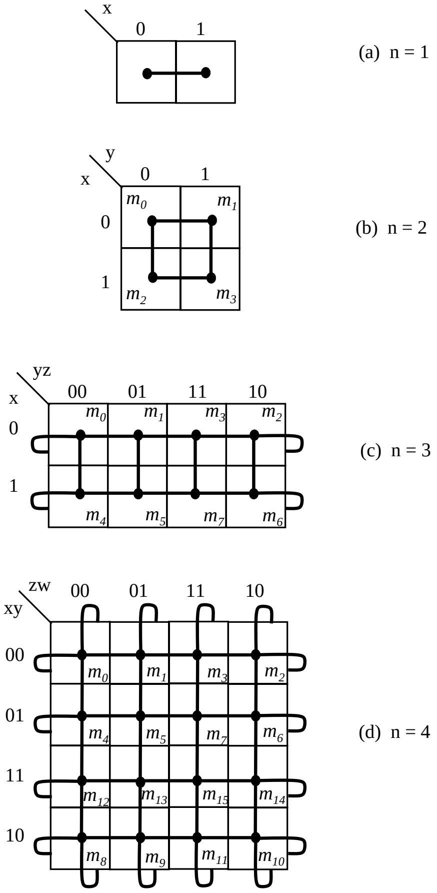


Figure 3: Boolean cubes and corresponding Karnaugh maps [Gajski].

- Each K-map defines a unique Boolean function.
  - A Boolean function can be represented by a truth table, a Boolean expression, an n-cube, or a map.

☞ K-map is in fact a visual diagram of all possible ways a function may be expressed—the simplest one can easily be identified.

- $\boxed{\text{K-maps provide visual aid to identify PIs and EPIs.}}$
- They are used for manual minimization of Boolean functions.

Exercise 1
How do you transform a K-map into a truth table? Is it unique? How do you transform a K-map into an n-cube? Is it unique? □

Two- and Three-Variable Maps

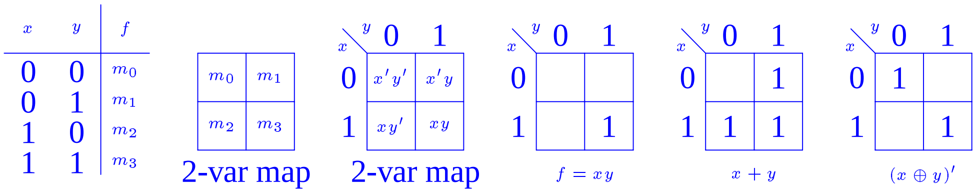


Figure 4: Two-variable maps.

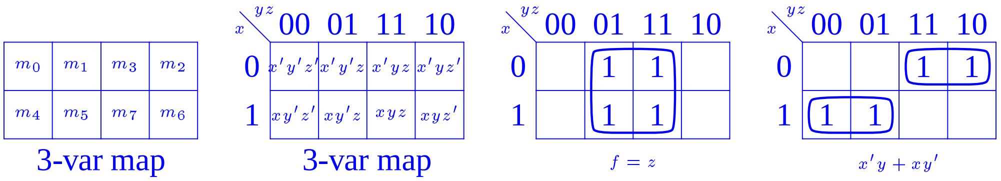
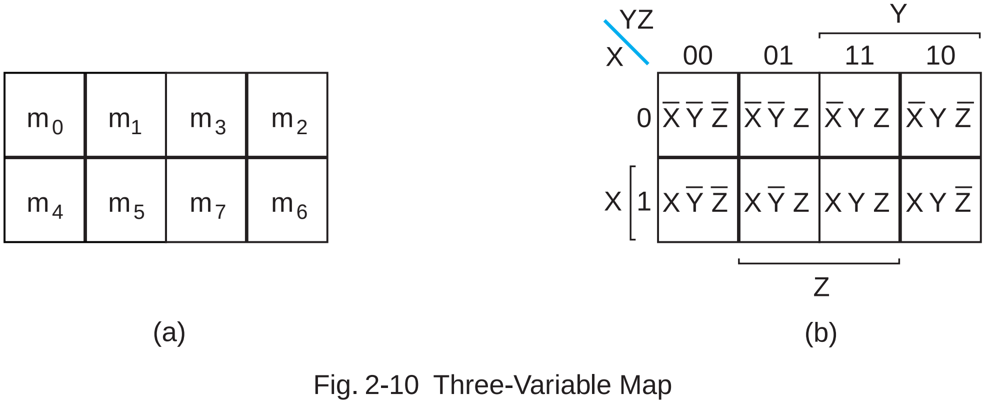


Figure 5: Three-variable maps.



m0 m1 m3 m2
m4 m5 m7 m6
YZ
X
Y
00 01 11 10
0 X̄ Ȳ Z̄ X̄ Ȳ Z X̄ Y Z X̄ Y Z̄
X 1 X Ȳ Z̄ X Ȳ Z X Y Z X Y Z̄
Z
(a)
(b)
Fig. 2-10 Three-Variable Map

Figure 6: Three-variable map simplification [Mano & Kime].

- Minterms are arranged in the Gray-code sequence. (Why?)
- Any 2 (horizontally or vertically) adjacent squares differ by exactly 1 variable, which is complemented in one square and uncomplemented in the other.
- Any 2 minterms in adjacent squares that are ORed together will cause a removal of the different variable, e.g., $m_5 + m_7 = xy'z + xyz = xz(y' + y) = xz$, because $y + y' = 1$.

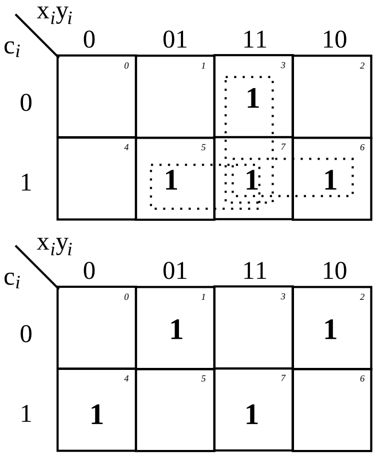
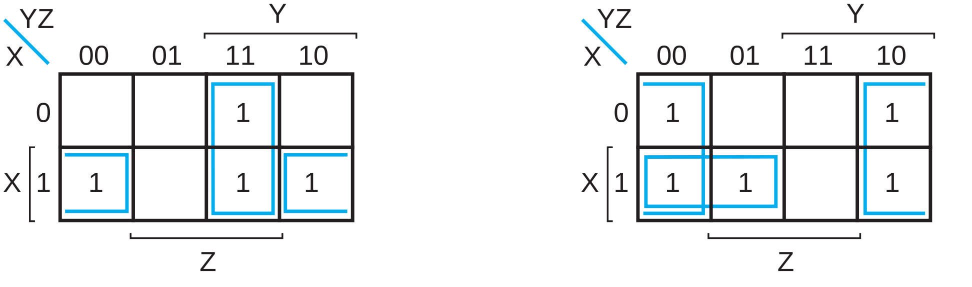


Figure 7: K-maps for the carry and sum functions [Gajski].

Example 2

Simplify the following Boolean functions: (a) $F_1(X, Y, Z) = \sum(3, 4, 6, 7)$; (b) $F_2(X, Y, Z) = \sum(0, 2, 4, 5, 6)$.



(a) $F_1(X, Y, Z) = \Sigma m(3, 4, 6, 7)$
$= YZ + X\overline{Z}$

(b) $F_2(X, Y, Z) = \Sigma m(0, 2, 4, 5, 6)$
$= \overline{Z} + X\overline{Y}$

Fig. 2-14 Maps for Example 2-4

□

Example 3

Simplify the Boolean function $F(X,Y,Z) = \sum(1,3,4,5,6)$.

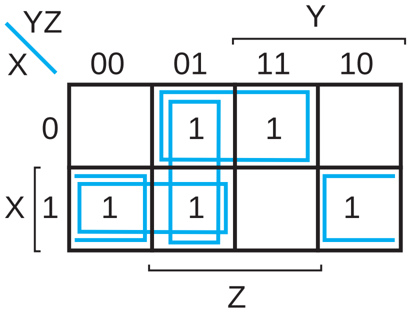


Fig. 2-15 $F(X,Y,Z) = \Sigma m(1, 3, 4, 5, 6)$

□

Example 4

Simplify the Boolean function $F = X'Z + X'Y + XY'Z + YZ$.

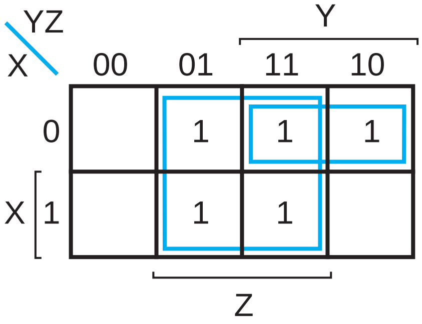


Fig. 2-16 $F(X,Y,Z) = \Sigma m\ (1,2,3,5,7)$

□

Four-Variable Maps

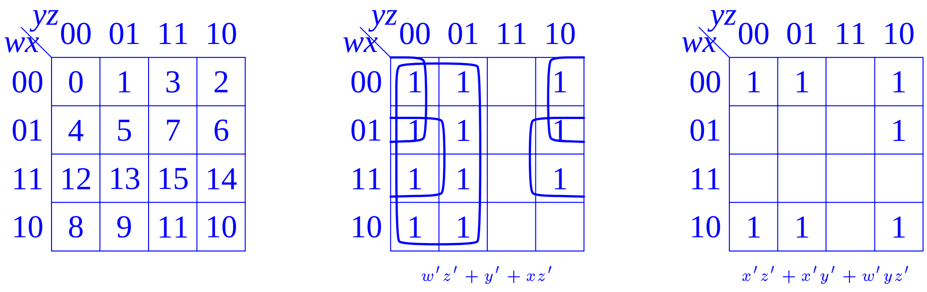
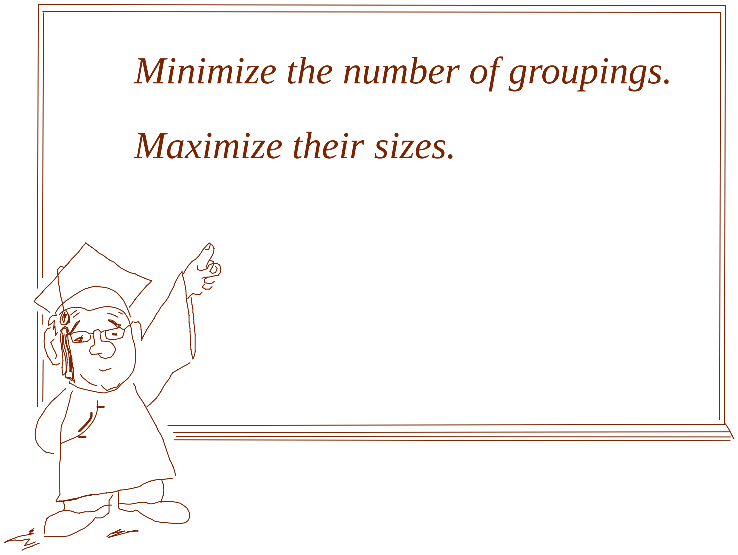


Figure 8: Four-variable maps.

- The map is considered to lie on a surface with the top and bottom edges, as well as the right and left edges, touching each other to form adjacent squares.
    - One square $\Rightarrow$ a minterm of 4 literals.
    - Two adjacent squares $\Rightarrow$ a term of 3 literals.
    - Four adjacent squares $\Rightarrow$ a term of 2 literals.
    - Eight adjacent squares $\Rightarrow$ a term of 1 literal.
    - Sixteen adjacent squares $\Rightarrow$ the constant ‘1’.

Example 5
For the maps shown above, $f_1(w, x, y, z) = \sum(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$ and $f_2 = w'x'y' + x'yz' + w'xyz' + wx'y'$. □

- A PI is a product term obtained by combining the maximum possible number of adjacent squares in the map.
- Pick the EPIs that minimize the number of literals.

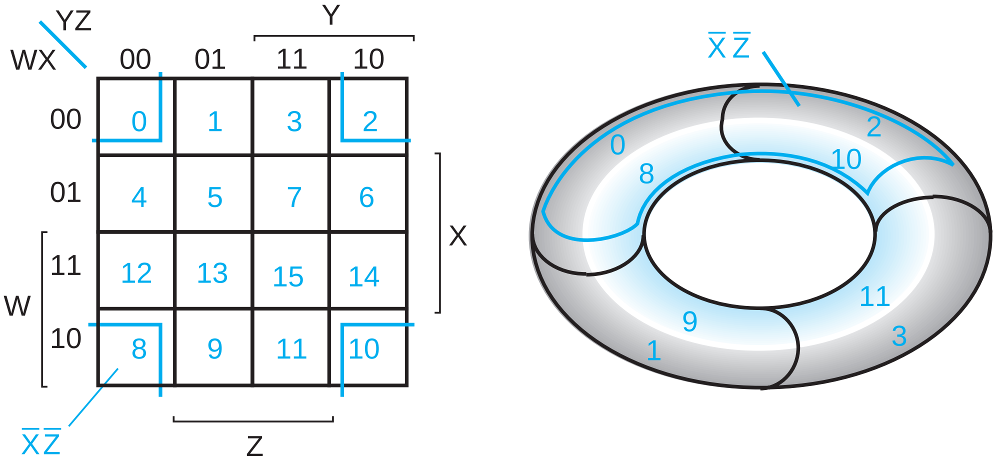


Minimize the number of groupings.

Maximize their sizes.

Example 6

Show the region in the K-map that is represented by $X'Z'$.



(a) (b)

Fig. 2-18 Four-Variable Map: Flat and on a Torus to Show Adjacencies

□

Example 7

Simplify the Boolean function $F(W, X, Y, Z) = \sum(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$.

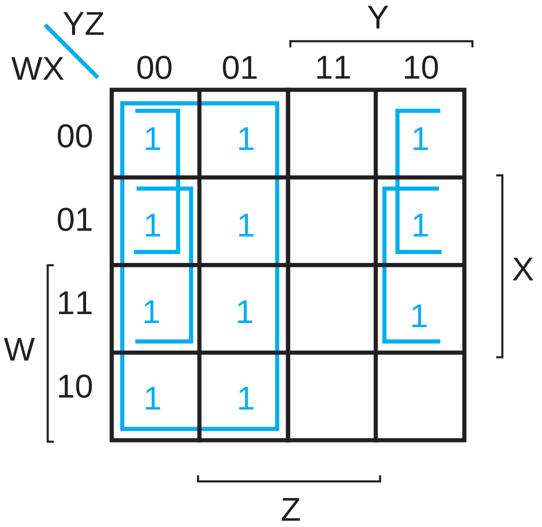


Fig. 2-19 Map for Example 2-5: $F = \overline{Y} + \overline{W}\,\overline{Z} + X\overline{Z}$

□

Example 8

Simplify the Boolean function $F = A'B'C' + B'CD' + AB'C' + A'BCD'$.

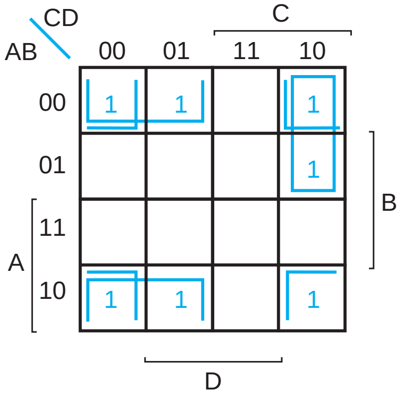


Fig. 2-20 Map for Example 2-6: $F = \bar{B}\bar{D} + \bar{B}\bar{C} + \bar{A}C\bar{D}$

□

*Five-Variable Maps

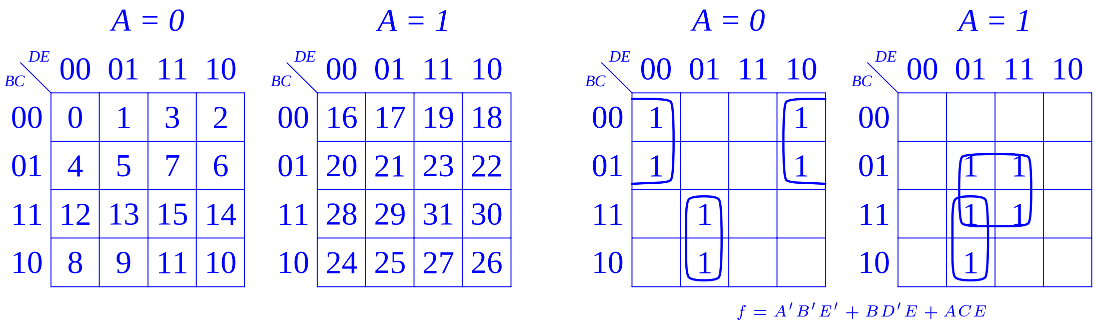


Figure 9: Five-variable maps.

- ☞ Imagine that the 2 maps are *superimposed* on one another.
- ☞ It is possible to construct a 6-variable map with four 4-variable maps by following a similar procedure.
- ☞ Maps of 6 or more variables are hard to read, and thus are impractical.

  ⟹ (1) Variable-entered maps (VEMs); (2) CAD programs.
- ☞ Any 2^k adjacent squares, $k = 0, 1, \ldots, n$, in an n-variable map represent an area that gives a product term of $n - k$ literals.

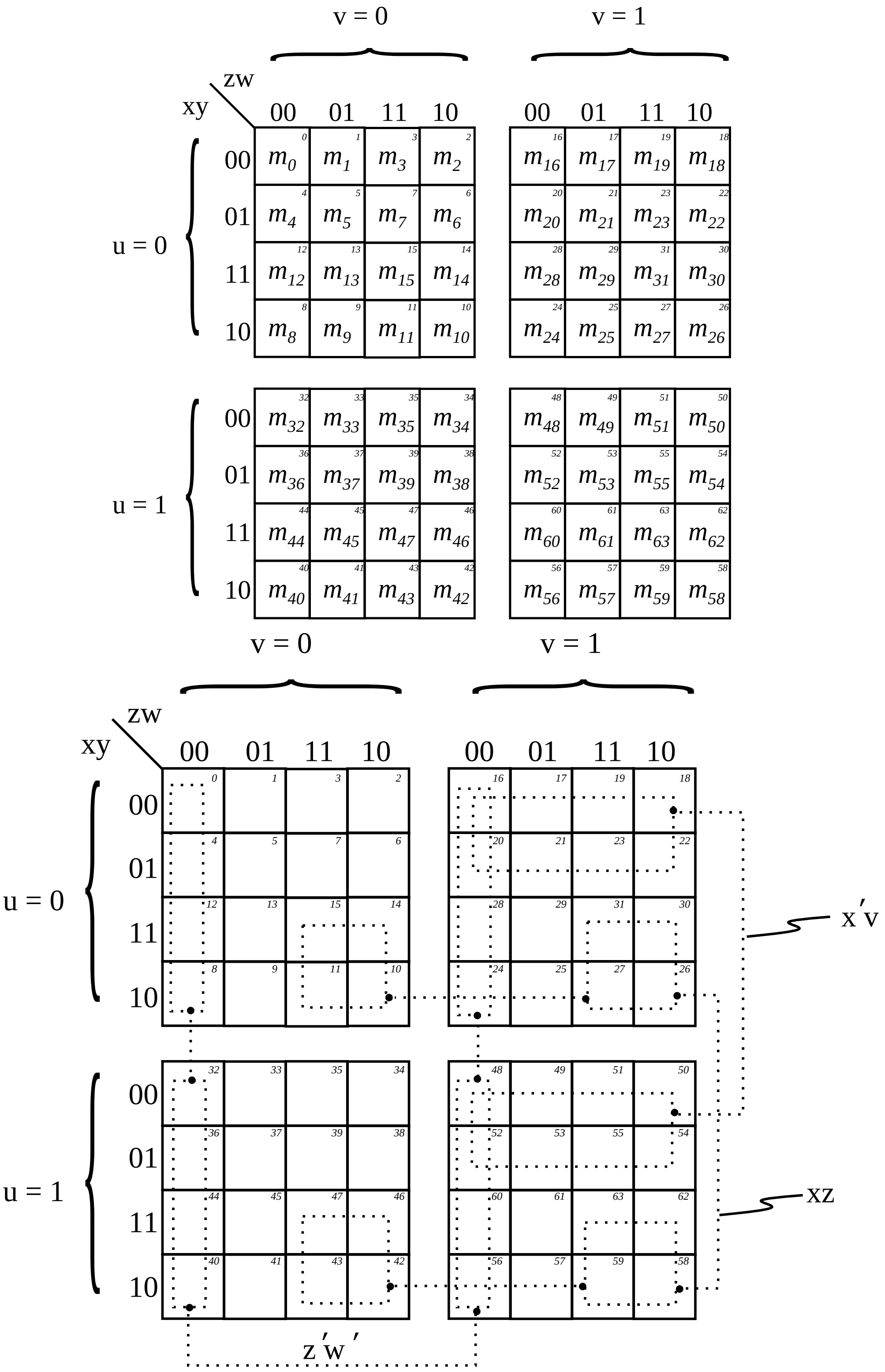


Figure 10: Six-variable K-maps [Gajski].

Theorem 1
In a K-map, 2 minterms are adjacent iff they differ in exactly 1 variable.

- In general, for an n-variable function, where $n = 2k$, we can construct a k-D K-map such that:
  - the map is extended in each dimension on 2 of the variables;
  - the extension sequence in each dimension is a Gray sequence.

Simplification Using K-Maps

- The complement of a function is represented in the map by the squares not marked by 1s (they usually are marked by 0s).

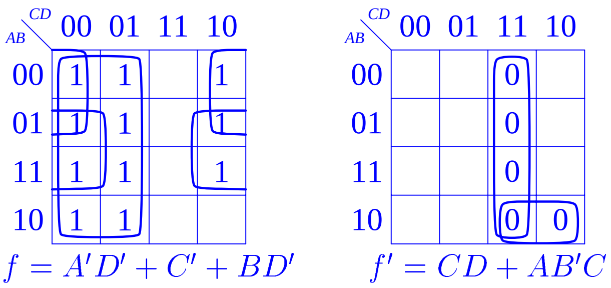


Figure 11: The complement of f' gives f in pos.

Example 9
We want to simplify the following function in sop & pos:

$$f(A, B, C, D) = \sum(0, 1, 2, 5, 8, 9, 10).$$

(a) Mark the map with 1s and 0s according to the function.

(b) Use the 1s to determine the EPIs of f, which immediately give the sop form:

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

(c) Use the 0s to determine the EPIs of f', and complement it to give the pos form:

$$f' = AB + CD + BD';$$

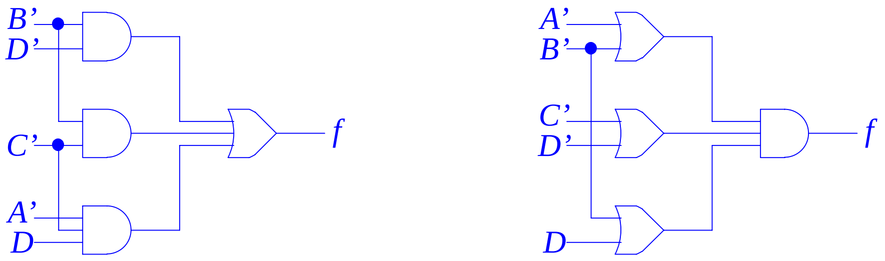


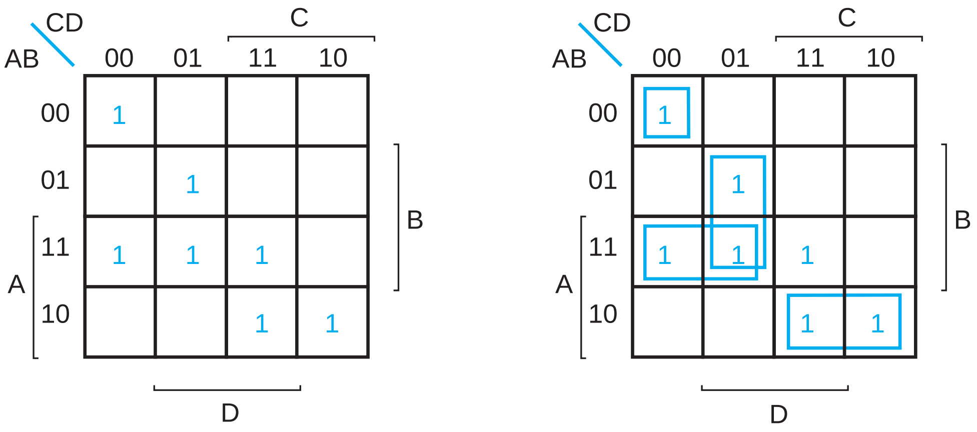
Figure 12: Two-level implementation.

$$f = (A' + B')(C' + D')(B' + D).$$

The 2-level (sop/pos) implementations using the AND/OR gates are shown below.

Note that NOT gates (inverters) are required if complemented inputs are not available. □

- The 1s of the function in the K-map or the truth table represent the minterms, and the 0s represent the maxterms.
- *Entering a function in the map*:
  - Entering a function expressed in sop in the map is straightforward.
  - To enter a function expressed in pos in the map, take the complement of the function (to get sop) and from it find the squares to be marked by 0s. The remaining squares are marked by 1s.
- *Simplification procedure*:
  - Obtain truth table, canonical form, or standard form.
  - Generate K-map.
  - Determine PIs.
  - Select EPIs.
  - Find minimal cover (set) of PIs.



(a) Plotting the minterms

(b) Essential prime implicants

Fig. 2-22 Simplification with Prime Implicants in Example 2-8

Figure 13: Simplification with PIs for $f = \sum(0, 5, 10, 11, 12, 13, 15)$ [Mano & Kime].

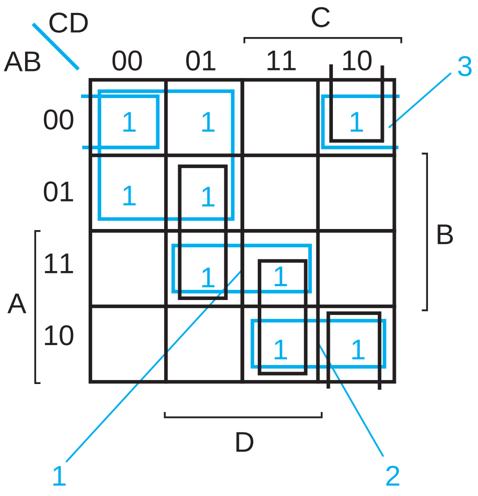


Fig. 2-23 Map for Example 2-9

Figure 14: Simplification with PIs for $f = \sum(0, 1, 2, 4, 5, 10, 11, 13, 15)$ [Mano & Kime].

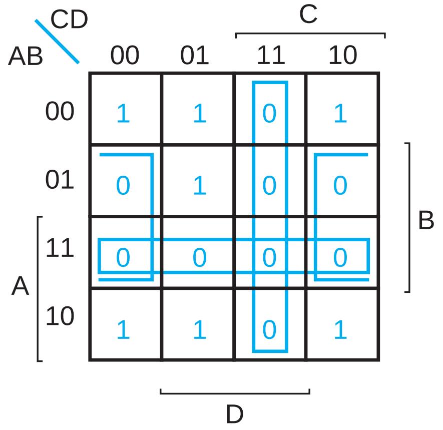
Exercise 2

Does the minimal cover of PIs contain only EPIs? □

Example 10

Simplify the following Boolean function in the pos form:

$F(A,B,C,D) = \sum(0,1,2,5,8,9,10)$.



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

Fig. 2-24 Map for Example 2-10: $F = (\bar{A} + \bar{B})(\bar{C} + \bar{D})(\bar{B} + D)$

$F' = AB + CD + BD' \;\therefore F = (A' + B')(C' + D')(B' + D)$ □

Exercise 3

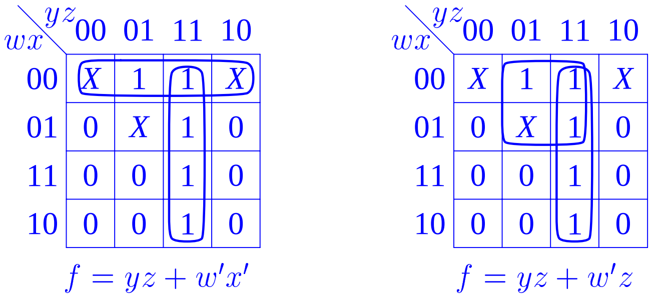
Simplify $F = (A' + B' + C)(B + D)$ in the pos form. □

Don't-Care Conditions

- In practice, there are applications where the function is not specified for certain combinations of the input variables. For example, in the 4-bit BCD code for the decimal digits, the outputs are unspecified for the input combinations 1010-1111.
- Functions that have unspecified outputs for some input combinations are called incompletely specified functions.
- The unspecified minterms of a function are called the don't-care conditions, or simply the don't-cares, and are denoted as Xs.
- These don't-care conditions can be used on a map to provide further simplification of the Boolean expression.

- Each X can be assigned an arbitrary value, 0 or 1, to help the simplification procedure.

Example 11
Simplify the Boolean function $f(w,x,y,z) = \sum(1,3,7,11,15)$ that has the don't-care conditions $d(w,x,y,z) = \sum(0,2,5)$.



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

$f = yz + w'x'$

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

$f = yz + w'z$

Figure 15: Simplification using don't-cares.

□

- Either one of the above expressions satisfies the conditions stated.
- Note that the above 2 expressions represent 2 functions that are algebraically unequal: each covers different don't-care terms.
- We may or may not include any of the Xs, while all the 1s must be included.
- It is also possible to obtain a simplified pos expression using the don't-cares.

Tabulation (Quine-McCluskey) Method

- The map method is convenient when the number of variables does not exceed 5 or 6.
- The map method is essentially a trial-and-error method that does not offer any guarantee of producing the best realization.
- The map method's dependence on the somewhat *intuitive* human ability to recognize patterns makes it unsuitable for design automation (programming for a digital computer).

- The tabulation method is a specific step-by-step (algorithmic) procedure that is guaranteed to produce a simplified standard-form expression for a function.
- It can be applied to problems with any number of variables.
- It is suitable for machine computation.
- It however is quite tedious for human use.
- The tabulation method was first formulated by Quine (1952) and later improved by McCluskey (1956), thus known as the Quine-McCluskey method.
  - Step 1: Exhaustive search of all PIs.
    - Group minterms by number of 1s.
    - Compare minterms and find pairs with distance 1.
    - Generate subcubes.
    - Repeat the above procedure on generated subcubes until no more subcubes can be generated.
  - Step 2: Choose among the PIs which give an expression with the least amount of literals (minimal cover generation).
    - Find EPIs through a selection table.
    - Find minimal cover through the pos of the PIs.

Determination of PIs

- The goal of the Quine-McCluskey algorithm is a special case of the following: *Select the smallest possible subset, D, of a set of objects, A, so that some criterion, f, is satisfied.*
- The set D will consist of all products in the minimal sop realization of a given Boolean function f, of which the set A contains all possible products.

Let $f = \sum(0, 1, 2, 8, 10, 11, 14, 15)$.

(a)			(b)			(c)	
	$wxyz$			$wxyz$			$wxyz$
0	0000	✓	0,1	000–		0,2,8,10	–0–0
			0,2	00–0	✓	0,8,2,10	–0–0
1	0001	✓	0,8	–000	✓		
2	0010	✓				10,11,14,15	1–1–
8	1000	✓	2,10	–010	✓	10,14,11,15	1–1–
			8,10	10–0	✓		
10	1010	✓					
			10,11	101–	✓		
11	1011	✓	10,14	1–10	✓		
14	1110	✓					
			11,15	1–11	✓		
15	1111	✓	14,15	111–	✓		

① Group the minterms according to the number of 1s (see Column (a)).

② Combine any 2 minterms that differ from each other by exactly one variable (i.e., dist-1), the unmatched variable removed (see Column (b)). Try this for all possible pairs of minterms. A check (✓) is placed to the right of both minterms if they have been used in a match.

③ Repeat the process. Combine any 2 product terms from Step ② that differ from each other by exactly one variable, the unmatched variable removed (see Column (c)).

④ The unchecked terms in the table form the PIs. Some of the product terms may appear twice in the table. It of course is unnecessary to use the same term twice.

☞ When comparing 2 terms to decide if they can be combined, the comparison can be done directly on the decimal numbers. We combine 2 terms iff the difference of their corresponding decimal numbers is a power of 2.

☞ In most cases, the sum of PIs obtained from the procedure shown above is not necessarily a standard form (which is minimized). The reason is that some of the PIs may be redundant.

Exercise 4
Use the map method to simplify $f = \sum(0, 1, 2, 8, 10, 11, 14, 15)$, and compare the result with the tabulation method. □

Minimal Cover Generation

The selection of PIs that form the minimized function is made from a PI table: each PI is represented in a row and each minterm in a column.

Suppose we have the following PIs: $x'y'z$, $w'xz'$, $w'xy$, xyz, wyz, and wx'.

	PI	Minterms	1	4	6	7	8	9	10	11	15
✓	$x'y'z$	1,9	X					X			
✓	$w'xz'$	4,6		X	X						
	$w'xy$	6,7			X	X					
	xyz	7,15				X					X
	wyz	11,15								X	X
✓	wx'	8,9,10,11					X	X	X	X	
			✓	✓	✓		✓	✓	✓	✓	

① Check the PIs that cover minterms with a single X in their columns. These PIs are the EPIs.

② Check each column whose minterm is covered by the selected EPIs.

③ If there are minterms left uncovered (7 & 15 in this example), some non-essential PIs have to be selected to cover them. We of course will select the PIs with a smallest total number of literals (xyz in this example).

*Multi-Output Minimization Using Maps

- Identify all possible PIs that cover each implicated minterms in each output expression, and search for a minimal cover by using *shared* terms.
- The Quine-McCluskey method also can be extended for this purpose.

$$\begin{aligned} f_1(A,B,C) &= \sum(0,2,3,5,6) \\ f_2(A,B,C) &= \sum(1,2,3,4,7) \\ f_3(A,B,C) &= \sum(2,3,4,5,6) \end{aligned}$$

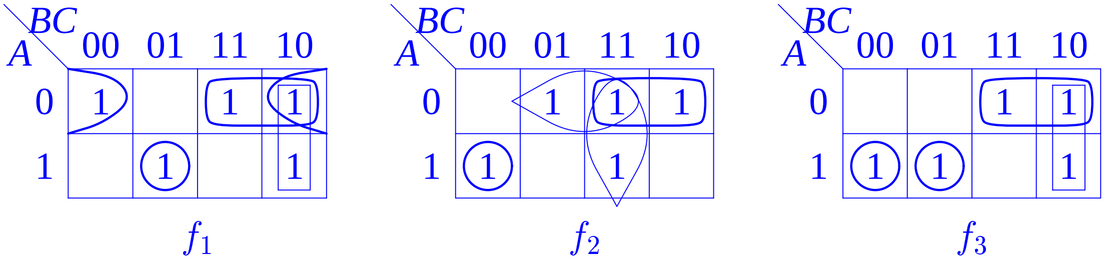
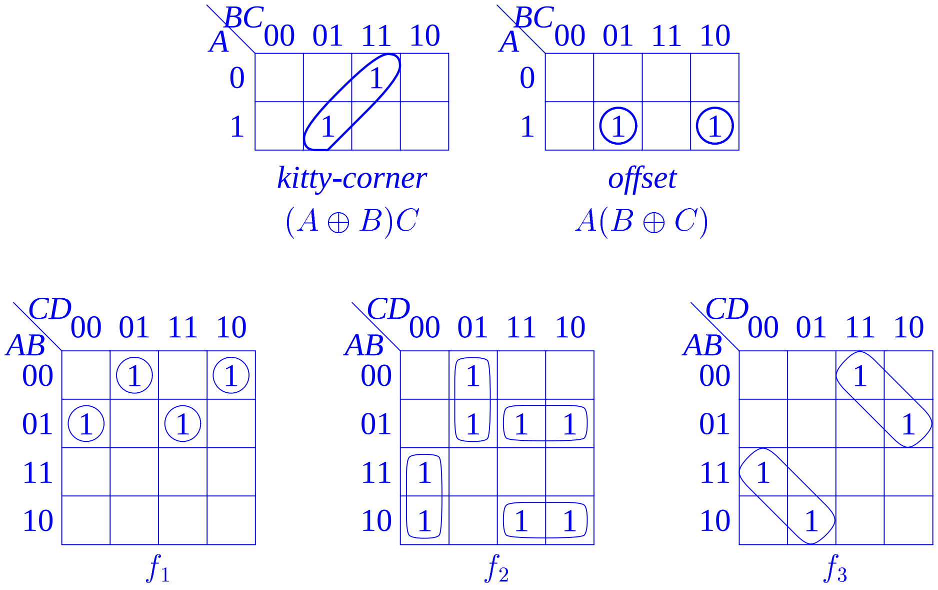


Figure 16: Simplification of multiple output functions.

$$\begin{aligned} f_1 &= A'C' + BC' + A'B + AB'C \\ f_2 &= A'C + BC + AB'C' + A'B \\ f_3 &= BC' + A'B + AB'C' + AB'C \end{aligned}$$

*XOR & XNOR Patterns on the Map

☞ Inspect the K-map to detect XOR/XNOR patterns: 1) kitty-corner adjacencies; 2) offset adjacencies.



kitty-corner
$(A \oplus B)C$

offset
$A(B \oplus C)$

f_1 f_2 f_3

Figure 17: XOR/XNOR patterns on the map.

$$\begin{aligned}
f_1 &= A'B'C'D + A'B'CD' + A'BC'D' + A'BCD \\
&= A'B'(C \oplus D) + A'B(C \oplus D)' \\
&= A'[(B')(C \oplus D) + (B)(C \oplus D)'] \\
&= A'[B \oplus (C \oplus D)]
\end{aligned}$$

$$\begin{aligned}
f_2 &= A'C'D + AC'D' + A'BC + AB'C \\
&= C'(A \oplus D) + C(A \oplus B)
\end{aligned}$$

$$\begin{aligned}
f_3 &= A'B'CD + A'BCD' + ABC'D' + AB'C'D \\
&= (A \oplus C)(B \oplus D)
\end{aligned}$$

Two-Level Implementations

- Digital circuits are more frequently constructed with NAND/NOR gates than with AND/OR/NOT gates due to ease of fabrication. For example, in gate arrays, only NAND (or NOR) gates are used.
- The conversion process from an expression/schematic with AND, OR, and NOT gates to one with only NAND or NOR gates is an example of technology mapping.

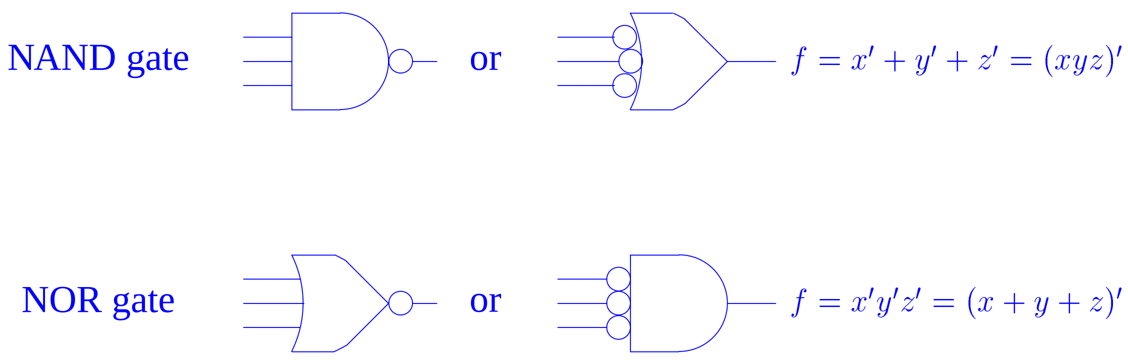


Figure 18: NAND/NOR gates.

- *NAND-NAND implementation*:
  - Simplify the function and express it in **sop**:

$$f = AB + CD + E.$$

  - Transfer it to 2-level NAND-NAND expression (DeMorgan's Thm):

$$f = ((AB + CD + E)')' = ((AB)'(CD)'E')'.$$

  - Draw the corresponding NAND gate implementation. Note that a 1-input NAND gate can replace an inverter.

- AND-OR ≡ NAND-NAND
- The process can be done directly on the logic diagram.

- *NOR-NOR implementation*:
  - Simplify the function and express it in **pos**:

    $$f = (A + B)(C + D)E.$$

  - Transfer it to 2-level NOR-NOR expression (DeMorgan's Thm):

    $$f = (((A + B)(C + D)E)')' = ((A + B)' + (C + D)' + E')'.$$

  - Draw the corresponding NOR gate implementation. Note that a 1-input NOR gate can replace an inverter.

- It is the dual of the NAND-NAND implementation.
- OR-AND ≡ NOR-NOR
- The process can be done directly on the logic diagram.
- The types of gates most often found in ICs are NAND & NOR, so NAND & NOR logic implementations are the most important from a practical point of view.

- *NAND-AND & AND-NOR implementations*:
  - Both perform the AND-OR-INVERT (AOI) function.
- *OR-NAND & NOR-OR implementations*:
  - Both perform the OR-AND-INVERT (OAI) function.

- Because of the INVERT part in each case, it is convenient to use the simplification of f' (the complement of the function) instead of f.

$x \backslash yz$	00	01	11	10
0	1	0	0	0
1	0	0	0	1

$f = x'y'z' + xyz'$

$f' = x'y + xy' + z$

Figure 19: NAND/NOR gates.

Example 12

Consider the function f given in Fig. 19.

AND-OR: $f = x'y'z' + xyz'$.
NAND-NAND: $f = (f')' = [(x'y'z')'(xyz')']'$.
OR-NAND: $f = [(x + y + z)(x' + y' + z)]'$.
NOR-OR: $f = (x + y + z)' + (x' + y' + z)'$.
AND-NOR: $f = (f')' = (x'y + xy' + z)'$.
NAND-AND: $f = (x'y)'(xy')'z'$.
OR-AND: $f = (x + y')(x' + y)z'$.
NOR-NOR: $f = (f')' = [(x + y')' + (x' + y)' + (z')']'$. □

Example 13

Implement $F(X, Y, Z) = \sum(1, 2, 3, 4, 5, 7)$ with NAND gates.

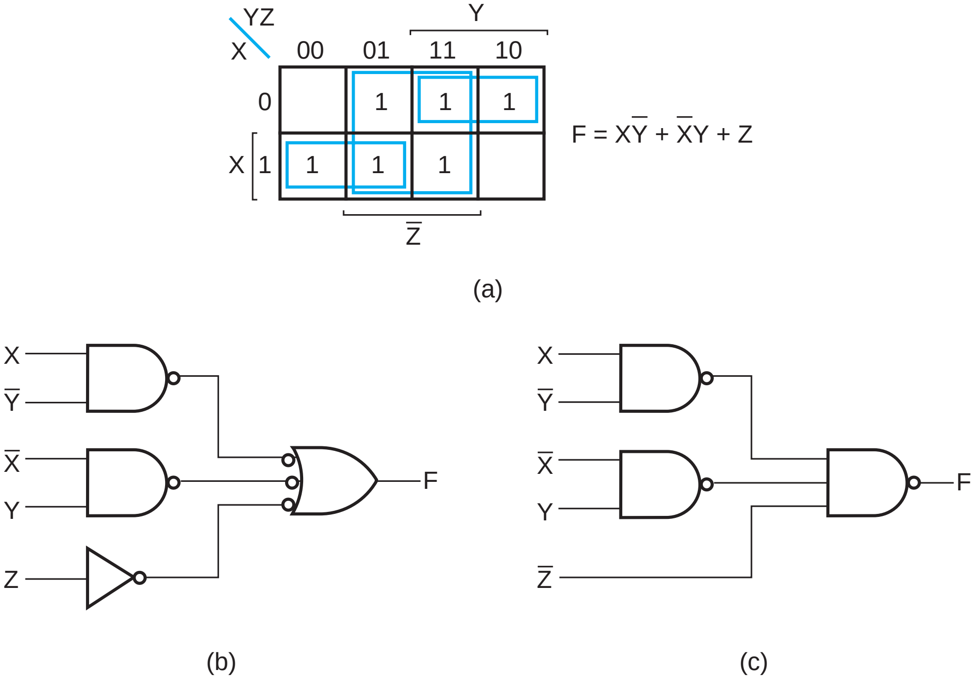
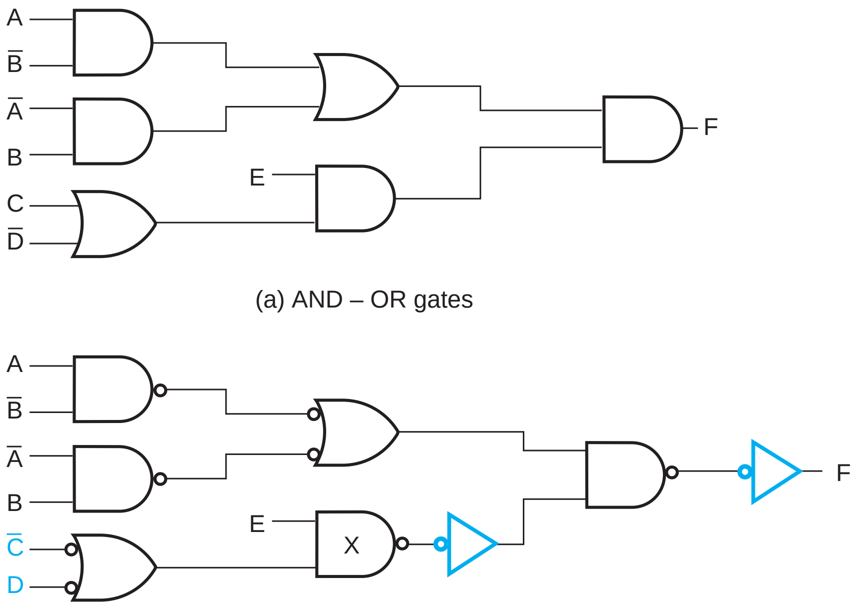


Fig. 2-30 Solution to Example 2-12

□

Example 14

Implement $F = (AB' + A'B)(E(C + D'))$ with NOR gates.



(a) AND – OR gates

(b) NAND gates

Fig. 2-32 Implementing $F = (A\overline{B} + \overline{A}B)E(C + \overline{D})$

□

Exclusive-OR (XOR) Gates

☞ Some identities for XOR:

- ☆ $X \oplus 0 = X$; $X \oplus 1 = X'$.
- ☆ $X \oplus X = 0$; $X \oplus X' = 1$.
- ☆ $X \oplus Y' = (X \oplus Y)'$; $X' \oplus Y = (X \oplus Y)'$.

☞ The XOR function is both commutative and associative.

- ☆ $A \oplus B = B \oplus A$.
- ☆ $(A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C$.

Exercise 5

(a) How do you implement the Even function with XOR (XNOR) gates?

(b) Design an n-bit even-parity generator. Calculate the hardware cost (in terms of primitive gates) and performance (in terms of primitive-gate delays). □

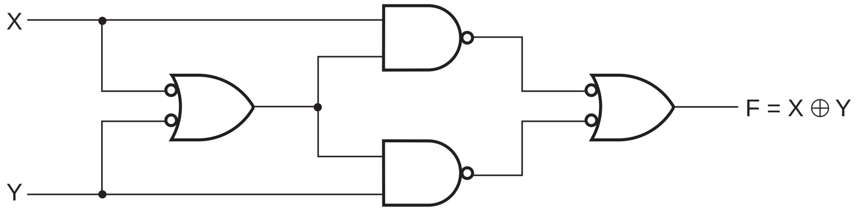
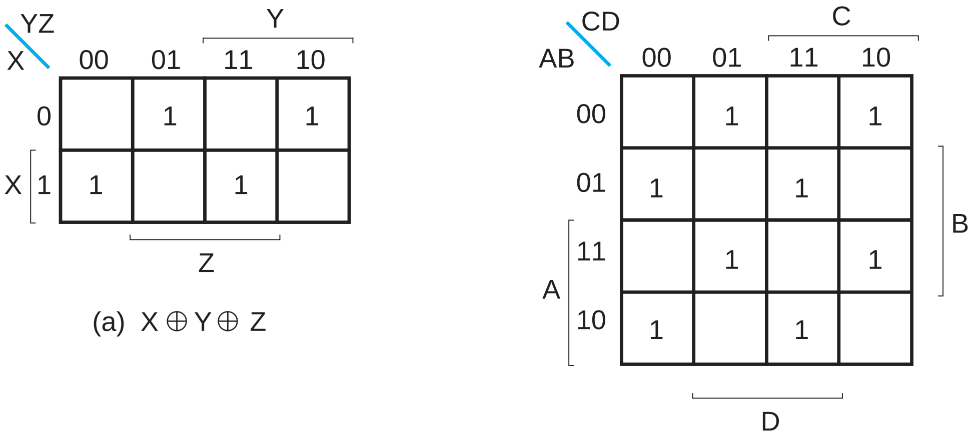


Fig. 2-37 Exclusive-OR Constructed with NAND Gates

Figure 20: Implementing the XOR function with NAND gates [Mano & Kime].



X \ YZ	00	01	11	10
0		1		1
1	1		1	

(a) X ⊕ Y ⊕ Z

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

(b) A ⊕ B ⊕ C ⊕ D

Fig. 2-38 Maps for Multiple-Variable Odd Functions

Figure 21: Implementing the Odd function with XOR gates [Mano & Kime].

*Technology Mapping

- A gate array is a 2-dimensional array of cells within which each cell contains a single NAND (NOR) gate that has a fixed number (usually 3) of inputs.
- The conversion process from an expression/schematic with AND, OR, and NOT gates to one with only NAND or NOR gates is an example of technology mapping.

$$
\begin{aligned}
&\text{Rule 1 :} && xy = ((xy)')' \\
&\text{Rule 2 :} && x + y = ((x+y)')' = (x'y')' \\
&\text{Rule 3 :} && xy = ((xy)')' = (x' + y')' \\
&\text{Rule 4 :} && x + y = ((x+y)')'
\end{aligned}
$$

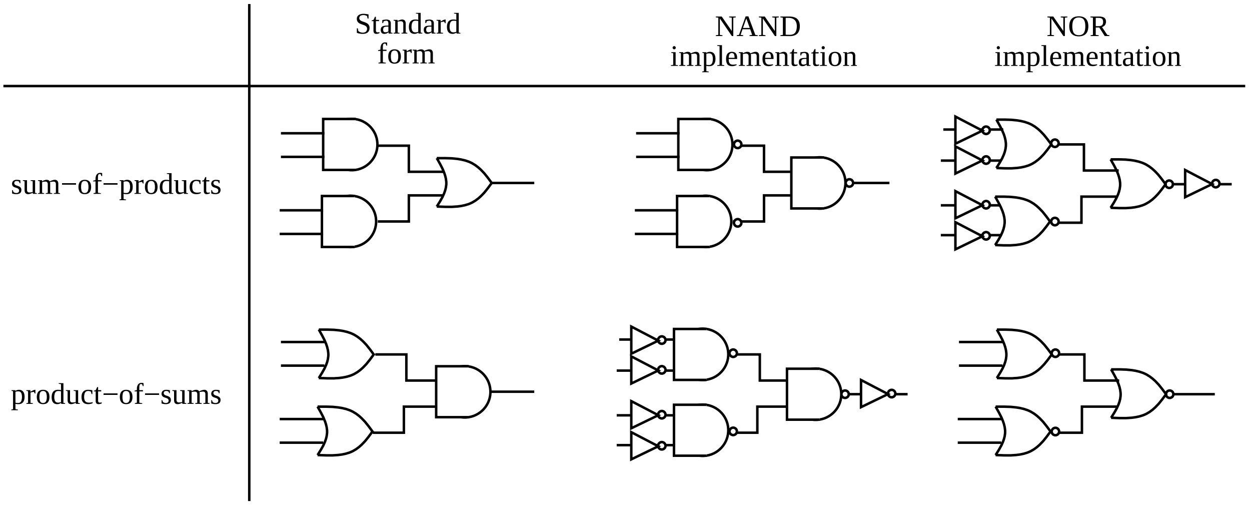


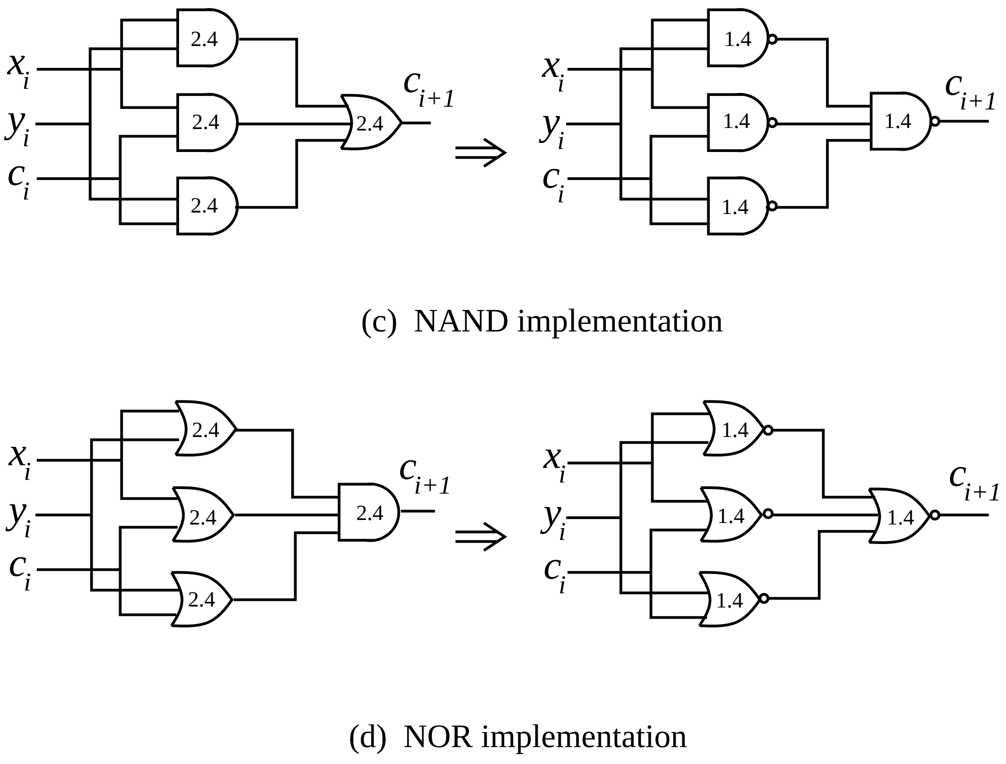
Figure 22: Conversion standard forms to NAND and NOR implementations [Gajski].

Example 15

Consider the carry function again:

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i = (x_i + y_i)(x_i + c_i)(y_i + c_i).$$

The NAND and NOR implementations of the carry function are shown in Fig 23.
□

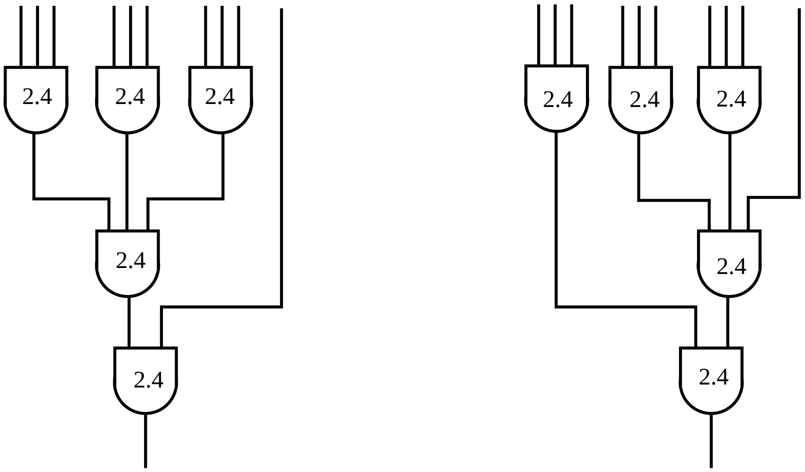


(c) NAND implementation

(d) NOR implementation

Figure 23: NAND and NOR implementations of the carry function [Gajski].

- ☞ Replace AND and OR gates with NAND gates by using Rules 1 and 2, and eliminate double inverters whenever possible.
- ☞ Replace AND and OR gates with NOR gates by using Rules 3 and 4, and eliminate double inverters whenever possible.
- ☞ Term decomposition: each n-input gate is decomposed into a tree of m-input gates, where $n > m$. For example, Fig 24 shows the decomposition of a 10-input AND gate into 3-input AND gates.
  - ⇒ The tree has $\lceil \log_m n \rceil$ levels and $\lceil (n-1)/(m-1) \rceil$ m-input gates.



(b) One possible decomposition (c) Alternative decomposition

Figure 24: Decomposition of a 10-input AND gate into 3-input AND gates [Gajski].

☞ Retiming: performance optimization (delay time minimization)—decomposition of a large gate can produce a tree in which different paths may incur different delays. For example, Fig 25 shows two NAND implementations of the 4-bit carry-look-ahead function:

$$c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0.$$

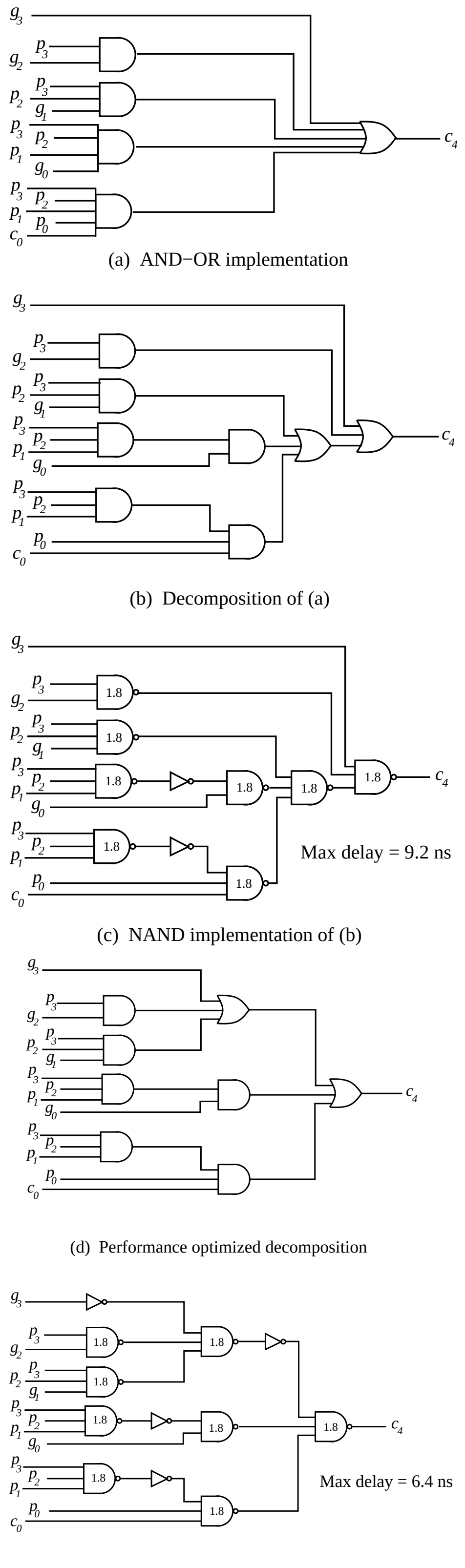
⇒ Carry generate function: $\boxed{g_i \equiv x_i y_i.}$

⇒ Carry propagate function: $\boxed{p_i \equiv x_i + y_i.}$

⇒ Carry look-ahead function: $\boxed{c_{i+1} = g_i + p_i c_i.}$

☞ Critical path: the delay on the longest input-output path.

☞ The rule of thumb is to minimize delay on critical paths and minimize cost on non-critical paths. A technology mapping example is shown in Fig 26.



(a) AND−OR implementation

(b) Decomposition of (a)

Max delay = 9.2 ns

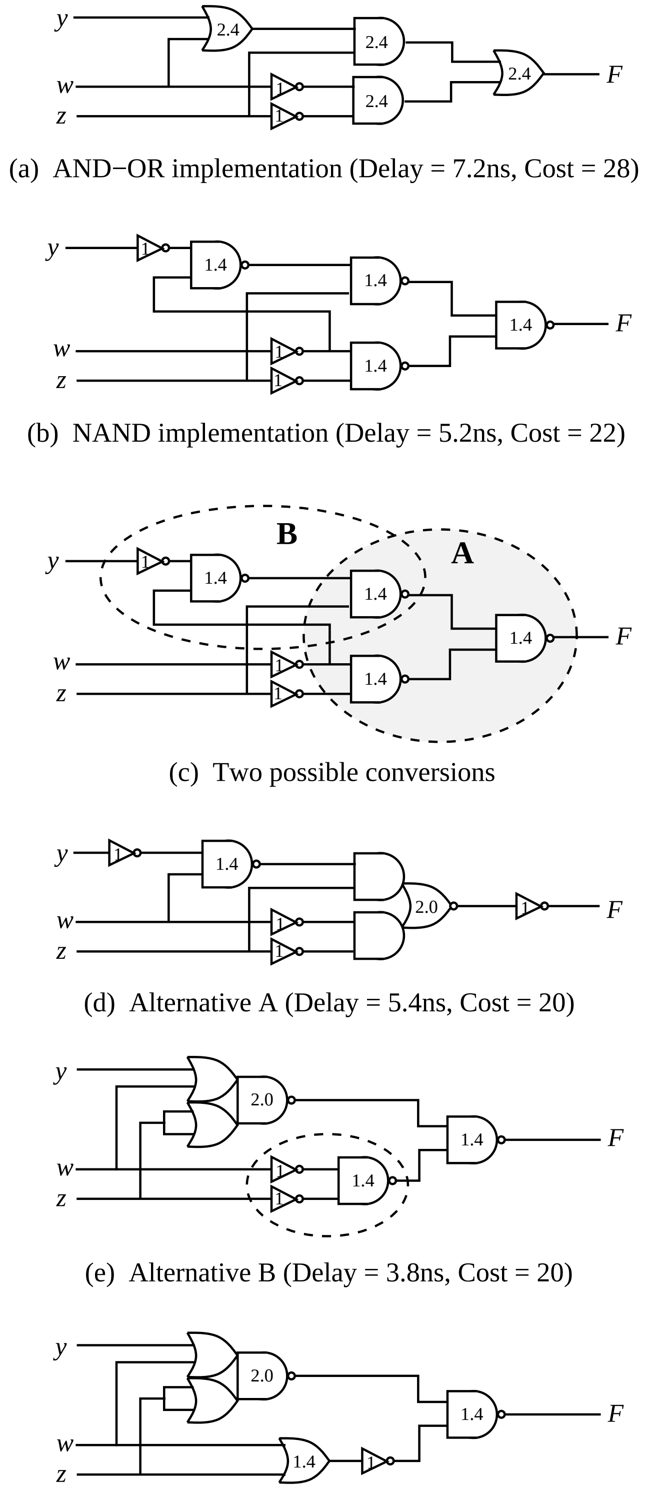
(c) NAND implementation of (b)

(d) Performance optimized decomposition

Max delay = 6.4 ns

(e) Performance optimized NAND implementation

Figure 25: Two NAND implementations of the 4-bit carry-look-ahead function [Gajski].



(a) AND−OR implementation (Delay = 7.2ns, Cost = 28)

(b) NAND implementation (Delay = 5.2ns, Cost = 22)

(c) Two possible conversions

(d) Alternative A (Delay = 5.4ns, Cost = 20)

(e) Alternative B (Delay = 3.8ns, Cost = 20)

(f) Cost optimized alternative B (Delay = 3.8ns, Cost = 18)

Figure 26: Technology mapping example [Gajski].

Hazard-Free Design

Definition 1
Hazards are unwanted switching transients (glitches) that appear at the output of a circuit because of different propagation delays on different but converging paths through the circuit.

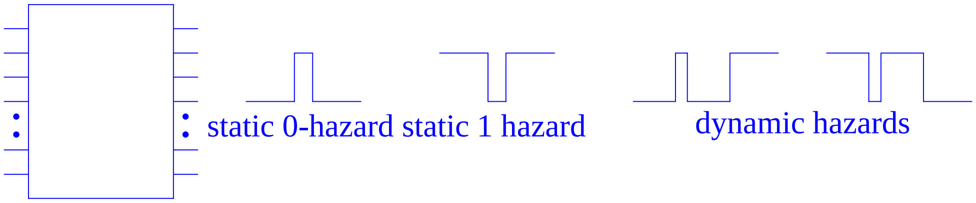
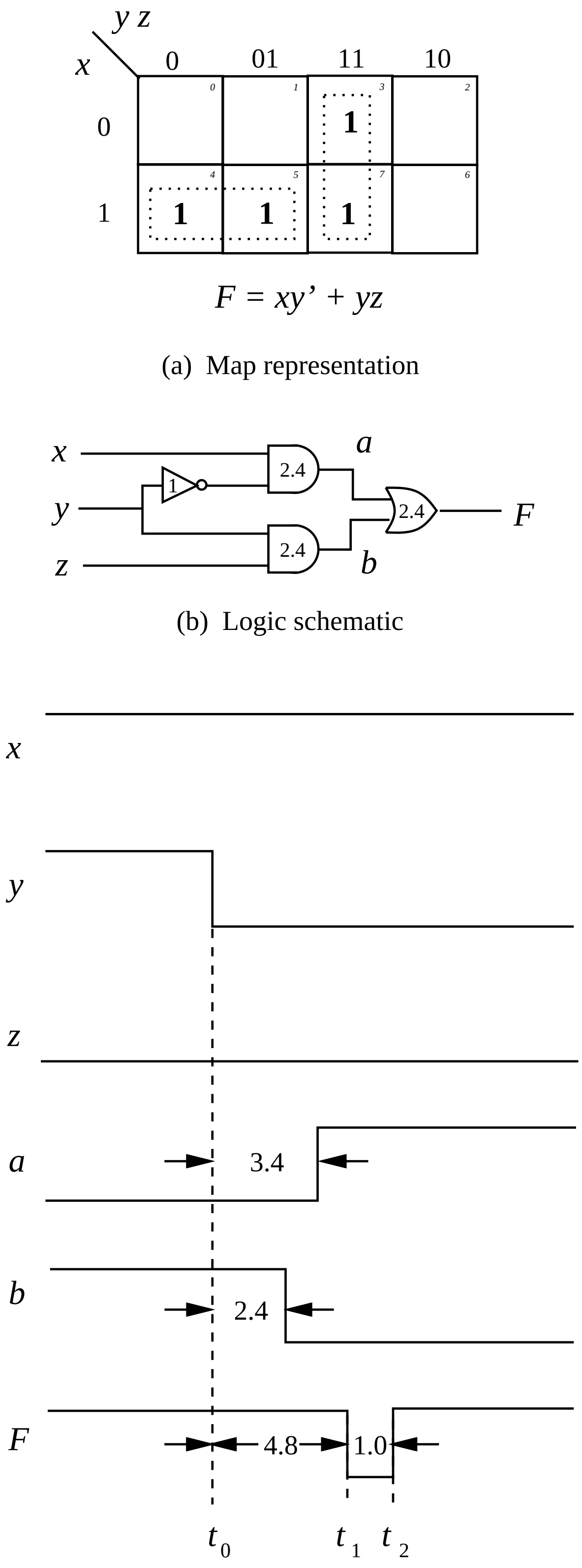


Figure 27: Hazards.

- Hazards can cause malfunction.
- Static-1 hazard: there are two 1-minterms that differ in only one variable but are not covered by a common product term in a sop implementation (Fig. 28).
  - It can be eliminated by including an additional PI covering both adjacent 1-minters (called the consensus term) (Fig. 29).
- Static-0 hazard: there are two 0-minterms that differ in only one variable but are not covered by a common sum term in a pos implementation.
- Static hazards are caused by two complementary signals which become equal for short periods of time due to different delays on different paths through the circuit.
- Dynamic hazard: two signals that always have the same value (even during the transition) become different for a short period of time.
  - It is a static hazard that occurs during the output transition.
  - It occurs when the same variable value propagates through the circuit on two different paths with different delays.
  - See Fig. 30 for an example.
    - It can be eliminated by introducing a redundant (consensus) PI.
    - It also can be eliminated by inserting a delay element.

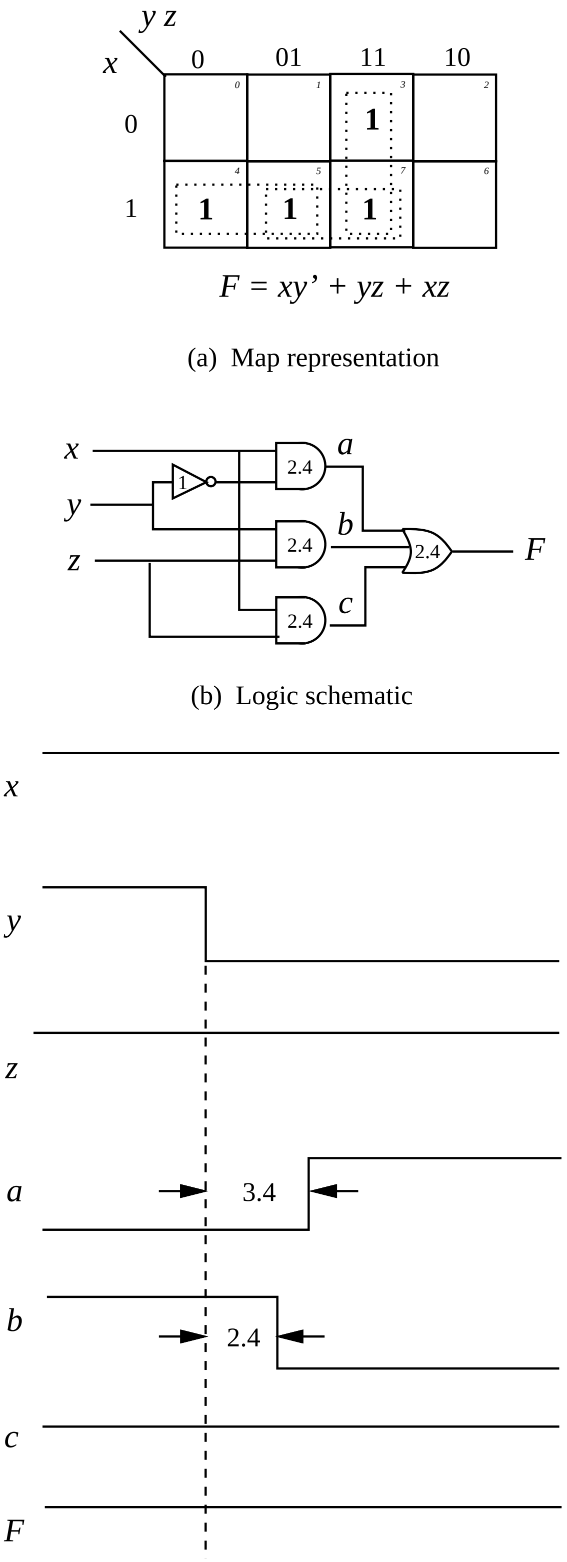


(a) Map representation

(b) Logic schematic

(c) Timing diagram

Figure 28: A design with static-1 hazard [Gajski].



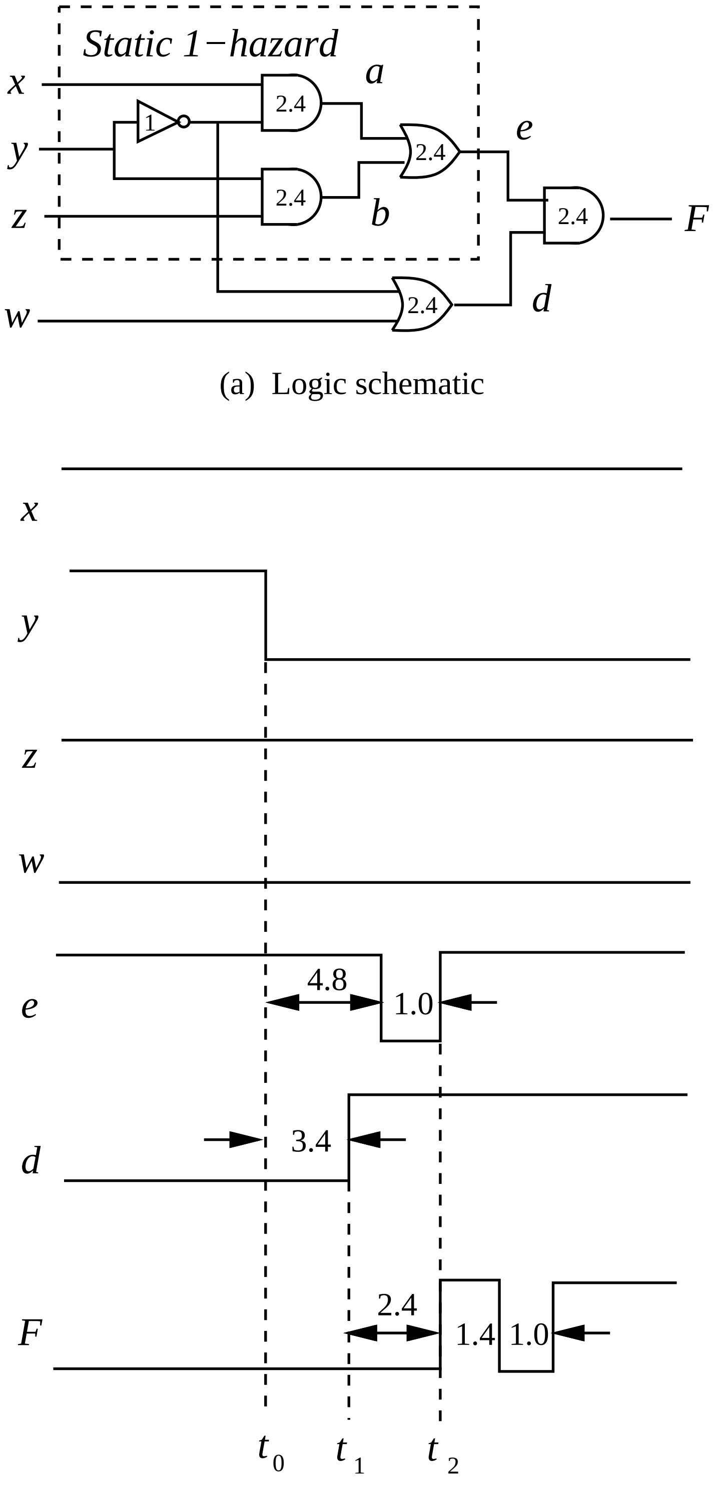
$F = xy' + yz + xz$

(a) Map representation

(b) Logic schematic

(c) Timing diagram

Figure 29: Hazard-free design [Gajski].



(a) Logic schematic

(b) Timing diagram

Figure 30: A design with dynamic hazard [Gajski].