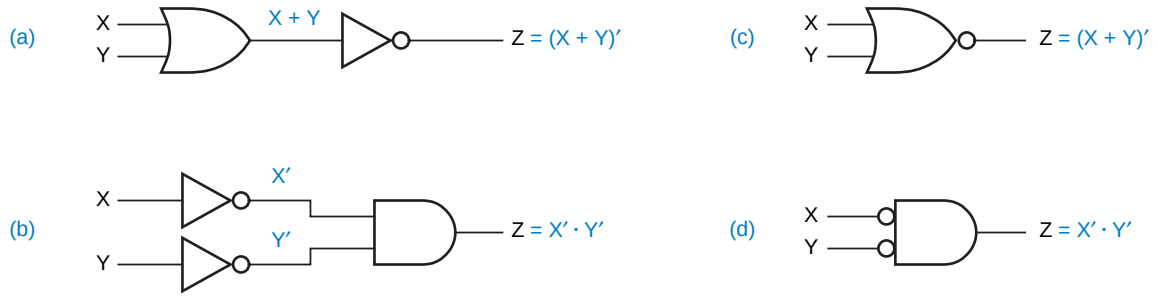


Figure 4-4 Equivalent circuits according to DeMorgan's theorem T13′: (a) OR-NOT; (b) NOT-AND; (c) logic symbol for a NOR gate; (d) equivalent symbol for a NOR gate.

A similar symbolic equivalence can be inferred from theorem T13′. As shown in Figure 4-4, a NOR gate may be realized as an OR gate followed by an inverter, or as inverters followed by an AND gate.

generalized DeMorgan's theorem

complement of a logic expression

Theorems T13 and T13′ are just special cases of a *generalized DeMorgan's theorem*, T14, that applies to an arbitrary logic expression F. By definition, the *complement of a logic expression*, denoted (F)′, is an expression whose value is the opposite of F's for every possible input combination. Theorem T14 is very important because it gives us a way to manipulate and simplify the complement of an expression.

Theorem T14 states that, given any *n*-variable logic expression, its complement can be obtained by swapping + and · and complementing all variables. For example, suppose that we have

$$\begin{aligned} F(W, X, Y, Z) &= (W' \cdot X) + (X \cdot Y) + (W \cdot (X' + Z')) \\ &= ((W)' \cdot X) + (X \cdot Y) + (W \cdot ((X)' + (Z)')) \end{aligned}$$

In the second line we have enclosed complemented variables in parentheses to remind you that the ′ is an operator, not part of the variable name. By applying theorem T14, we obtain

$$[F(W, X, Y, Z)]' = ((W')' + X') \cdot (X' + Y') \cdot (W' + ((X')' \cdot (Z')'))$$

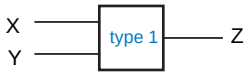
Using theorem T4, this can be simplified to

$$[F(W, X, Y, Z)]' = (W + X') \cdot (X' + Y') \cdot (W' + (X \cdot Z))$$

In general, we can use theorem T14 to complement a parenthesized expression by swapping + and ·, complementing all uncomplemented variables, and uncomplementing all complemented ones.

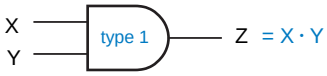
The generalized DeMorgan's theorem T14 can be proved by showing that all logic functions can be written as either a sum or a product of subfunctions,

(a)



X	Y	Z
LOW	LOW	LOW
LOW	HIGH	LOW
HIGH	LOW	LOW
HIGH	HIGH	HIGH

(b)



X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

(c)

X	Y	Z
1	1	1
1	0	1
0	1	1
0	0	0

Figure 4-5 A "type-1"logic gate: (a) electrical function table; (b) logic function table and symbol with positive logic; (c) logic function table and symbol with negative logic.

variables $X_1, X_2, \ldots, X_n$ and the operators $+$, $\cdot$, and $'$, then the dual of F, written F^D, is the same expression with $+$ and $\cdot$ swapped:

$$F^D(X_1, X_2, \ldots, X_n, +, \cdot, ') = F(X_1, X_2, \ldots, X_n, \cdot, +, ')$$

You already knew this, of course, but we wrote the definition in this way just to highlight the similarity between duality and the generalized DeMorgan's theorem T14, which may now be restated as follows:

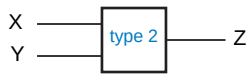
$$[F(X_1, X_2, \ldots, X_n)]' = F^D(X_1', X_2', \ldots, X_n')$$

Let's examine this statement in terms of a physical network.

Figure 4-5(a) shows the electrical function table for a logic element that we'll simply call a "type-1" gate. Under the positive-logic convention (LOW = 0 and HIGH = 1), this is an AND gate, but under the negative-logic convention (LOW = 1 and HIGH = 0), it is an OR gate, as shown in (b) and (c). We can also imagine a "type-2" gate, shown in Figure 4-6, that is a positive-logic OR or a negative-logic AND. Similar tables can be developed for gates with more than two inputs.

Figure 4-6 A "type-2" logic gate: (a) electrical function table; (b) logic function table and symbol with positive logic; (c) logic function table and symbol with negative logic.

(a)

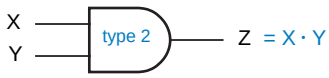


X	Y	Z
LOW	LOW	LOW
LOW	HIGH	HIGH
HIGH	LOW	HIGH
HIGH	HIGH	HIGH

(b)

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

(c)



X	Y	Z
1	1	1
1	0	0
0	1	0
0	0	0

Row	X	Y	Z	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

Table 4-7
Truth table for the logic circuit of Figure 4-9.

shown in Table 4-7. Once we have the truth table for the circuit, we can also directly write a logic expression—the canonical sum or product—if we wish.

The number of input combinations of a logic circuit grows exponentially with the number of inputs, so the exhaustive approach can quickly become exhausting. Instead, we normally use an algebraic approach whose complexity is more linearly proportional to the size of the circuit. The method is simple—we build up a parenthesized logic expression corresponding to the logic operators and structure of the circuit. We start at the circuit inputs and propagate expressions through gates toward the output. Using the theorems of switching algebra, we may simplify the expressions as we go, or we may defer all algebraic manipulations until an output expression is obtained.

Figure 4-11 applies the algebraic technique to our example circuit. The output function is given on the output of the final OR gate:

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

No switching-algebra theorems were used to obtain this expression. However, we can use theorems to transform this expression into another form. For example, a sum of products can be obtained by "multiplying out":

$$F = X \cdot Z + Y' \cdot Z + X' \cdot Y \cdot Z'$$

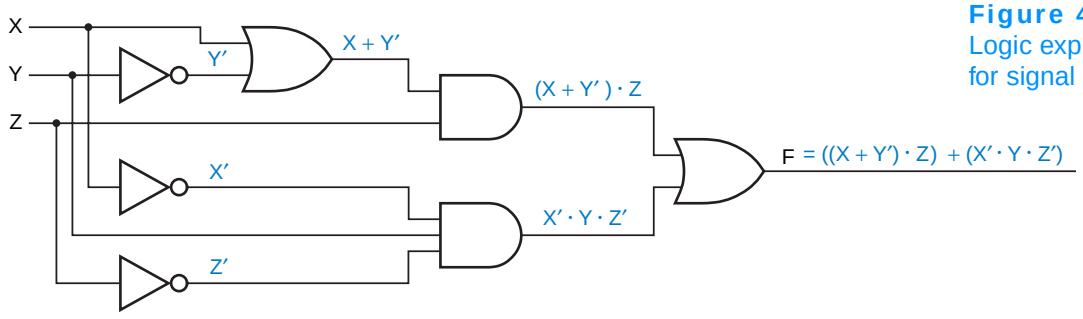


Figure 4-11
Logic expressions for signal lines.

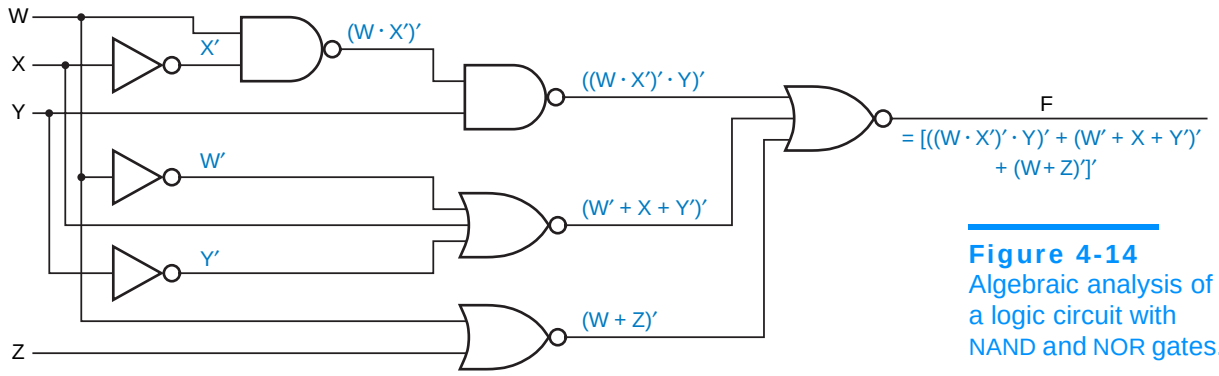


Figure 4-14
Algebraic analysis of a logic circuit with NAND and NOR gates.

$$
\begin{aligned}
F &= [((W \cdot X')' \cdot Y)' + (W' + X + Y')' + (W + Z)']' \\
&= ((W' + X)' + Y')' \cdot (W \cdot X' \cdot Y)' \cdot (W' \cdot Z')' \\
&= ((W \cdot X')' \cdot Y) \cdot (W' + X + Y') \cdot (W + Z) \\
&= ((W' + X) \cdot Y) \cdot (W' + X + Y') \cdot (W + Z)
\end{aligned}
$$

Quite often, DeMorgan's theorem can be applied *graphically* to simplify algebraic analysis. Recall from Figures 4-3 and 4-4 that NAND and NOR gates each have two equivalent symbols. By judiciously redrawing Figure 4-14, we make it possible to cancel out some of the inversions during the analysis by using theorem T4 [$(X')' = X$], as shown in Figure 4-15. This manipulation leads us to a simplified output expression directly:

$$F = ((W' + X) \cdot Y) \cdot (W' + X + Y') \cdot (W + Z)$$

Figures 4-14 and 4-15 were just two different ways of drawing the same physical logic circuit. However, when we simplify a logic expression using the theorems of switching algebra, we get an expression corresponding to a different physical circuit. For example, the simplified expression above corresponds to the circuit of Figure 4-16, which is physically different from the one in the previous two figures. Furthermore, we could multiply out and add out the

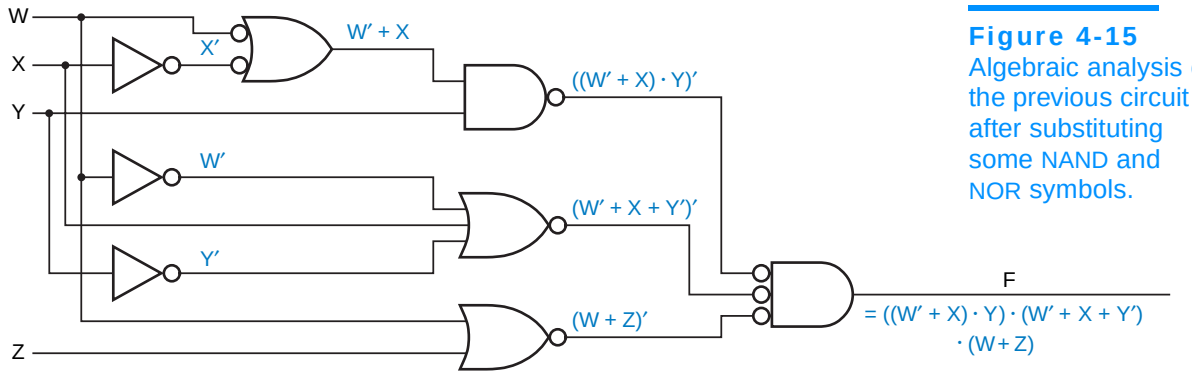


Figure 4-15
Algebraic analysis of the previous circuit after substituting some NAND and NOR symbols.

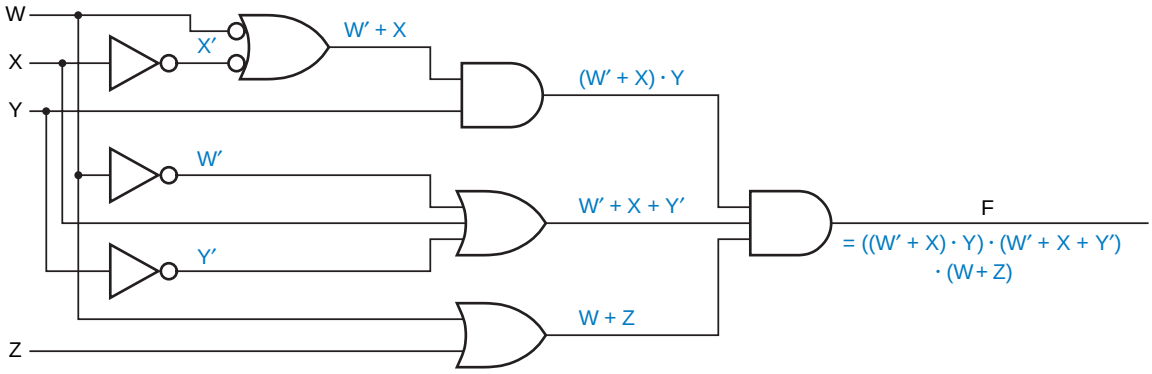
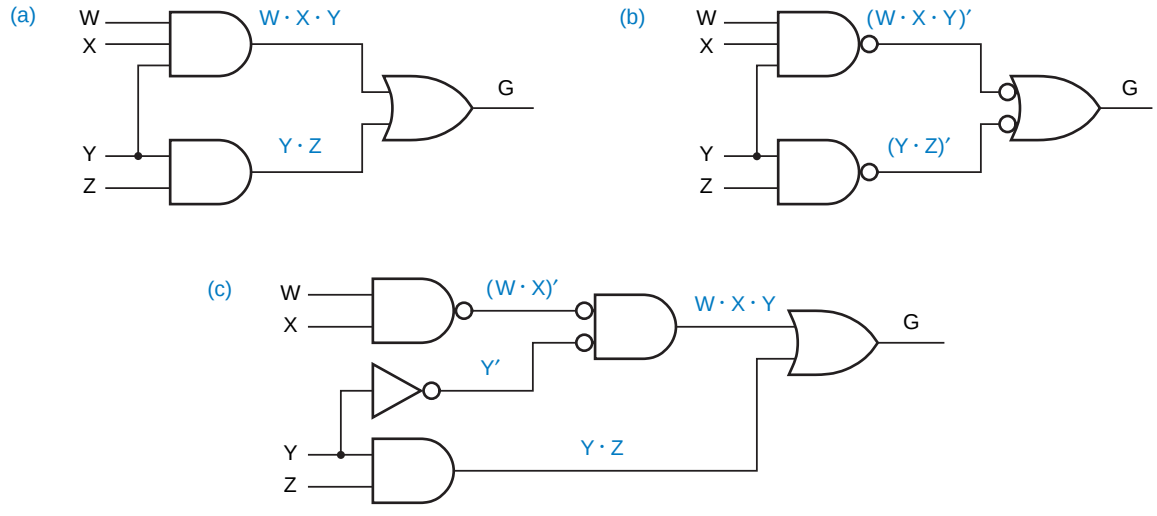


Figure 4-16 A different circuit for same logic function.

expression to obtain sum-of-products and product-of-sums expressions corresponding to two more physically different circuits for the same logic function.

Although we used logic expressions above to convey information about the physical structure of a circuit, we don't always do this. For example, we might use the expression G(W, X, Y, Z) = W · X · Y + Y · Z to describe any one of the circuits in Figure 4-17. Normally, the only sure way to determine a circuit's structure is to look at its schematic drawing. However, for certain restricted classes of circuits, structural information can be inferred from logic expressions. For example, the circuit in (a) could be described without reference to the drawing as "a two-level AND-OR circuit for W · X · Y + Y · Z," while the circuit in (b) could be described as "a two-level NAND-NAND circuit for W · X · Y + Y · Z."

Figure 4-17 Three circuits for G(W, X, Y, Z) = W · X ·Y + Y · Z: (a) two-level AND-OR; (b) two-level NAND-NAND; (c) ad hoc.



(a)

X	Y	Z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

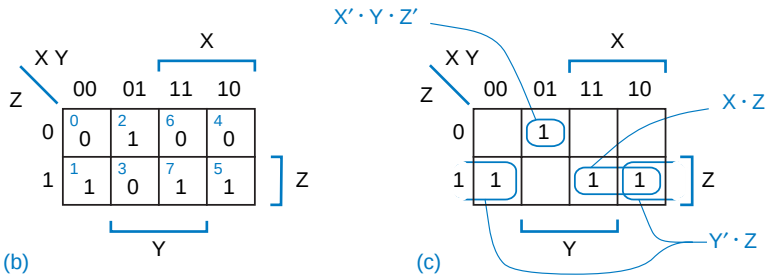


Figure 4-27 $F = \Sigma_{X,Y,Z}(1,2,5,7)$: (a) truth table; (b) Karnaugh map; (c) combining adjacent 1-cells.

4-variable maps, corresponding cells on the left/right or top/bottom borders are less obvious neighbors; for example, cells 12 and 14 in the 4-variable map are adjacent because they differ only in the value of Y.

Each input combination with a "1" in the truth table corresponds to a minterm in the logic function's canonical sum. Since pairs of adjacent "1" cells in the Karnaugh map have minterms that differ in only one variable, the minterm pairs can be combined into a single product term using the generalization of theorem T10, term · Y + term · Y′ = term. Thus, we can use a Karnaugh map to simplify the canonical sum of a logic function.

For example, consider cells 5 and 7 in Figure 4-27(b) and their contribution to the canonical sum for this function:

$$\begin{aligned} F &= \cdots + X \cdot Y' \cdot Z + X \cdot Y \cdot Z \\ &= \cdots + (X \cdot Z) \cdot Y' + (X \cdot Z) \cdot Y \\ &= \cdots + X \cdot Z \end{aligned}$$

Remembering wraparound, we see that cells 1 and 5 in Figure 4-27(b) are also adjacent and can be combined:

$$\begin{aligned} F &= X' \cdot Y' \cdot Z + X \cdot Y' \cdot Z + \cdots \\ &= X' \cdot (Y' \cdot Z) + X \cdot (Y' \cdot Z) + \cdots \\ &= Y' \cdot Z + \ldots \end{aligned}$$

In general, we can simplify a logic function by combining pairs of adjacent 1-cells (minterms) whenever possible and writing a sum of product terms that covers all of the 1-cells. Figure 4-27(c) shows the result for our example logic function. We circle a pair of 1s to indicate that the corresponding minterms are combined into a single product term. The corresponding AND-OR circuit is shown in Figure 4-28.

In many logic functions the cell-combining procedure can be extended to combine more than two 1-cells into a single product term. For example, consider

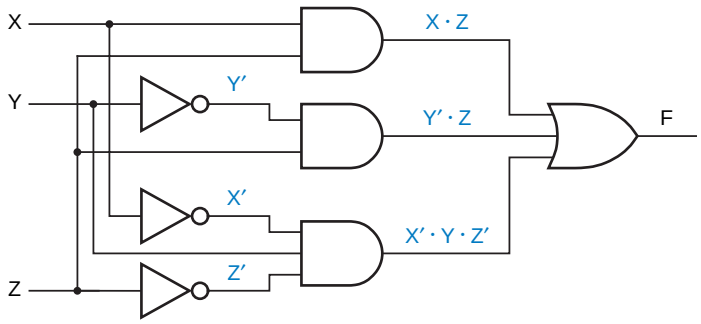


Figure 4-28
Minimized AND-OR circuit.

the canonical sum for the logic function $F = \Sigma_{X,Y,Z}(0, 1, 4, 5, 6)$. We can use the algebraic manipulations of the previous examples iteratively to combine four of the five minterms:

$$\begin{aligned} F &= X' \cdot Y' \cdot Z' + X' \cdot Y' \cdot Z + X \cdot Y' \cdot Z' + X \cdot Y' \cdot Z + X \cdot Y \cdot Z' \\ &= [(X' \cdot Y') \cdot Z' + (X' \cdot Y') \cdot Z] + [(X \cdot Y') \cdot Z' + (X \cdot Y') \cdot Z] + X \cdot Y \cdot Z' \\ &= X' \cdot Y' + X \cdot Y' + X \cdot Y \cdot Z' \\ &= [X' \cdot (Y') + X \cdot (Y')] + X \cdot Y \cdot Z' \\ &= Y' + X \cdot Y \cdot Z' \end{aligned}$$

In general, 2^i 1-cells may be combined to form a product term containing $n - i$ literals, where n is the number of variables in the function.

A precise mathematical rule determines how 1-cells may be combined and the form of the corresponding product term:

- A set of 2^i 1-cells may be combined if there are i variables of the logic function that take on all 2^i possible combinations within that set, while the remaining $n - i$ variables have the same value throughout that set. The corresponding product term has $n - i$ literals, where a variable is complemented if it appears as 0 in all of the 1-cells, and uncomplemented if it appears as 1.

rectangular sets of 1s

Graphically, this rule means that we can circle *rectangular* sets of 2^i 1s, literally as well as figuratively stretching the definition of rectangular to account for wraparound at the edges of the map. We can determine the literals of the corresponding product terms directly from the map; for each variable we make the following determination:

- If a circle covers only areas of the map where the variable is 0, then the variable is complemented in the product term.
- If a circle covers only areas of the map where the variable is 1, then the variable is uncomplemented in the product term.
- If a circle covers both areas of the map where the variable is 0 and areas where it is 1, then the variable does not appear in the product term.

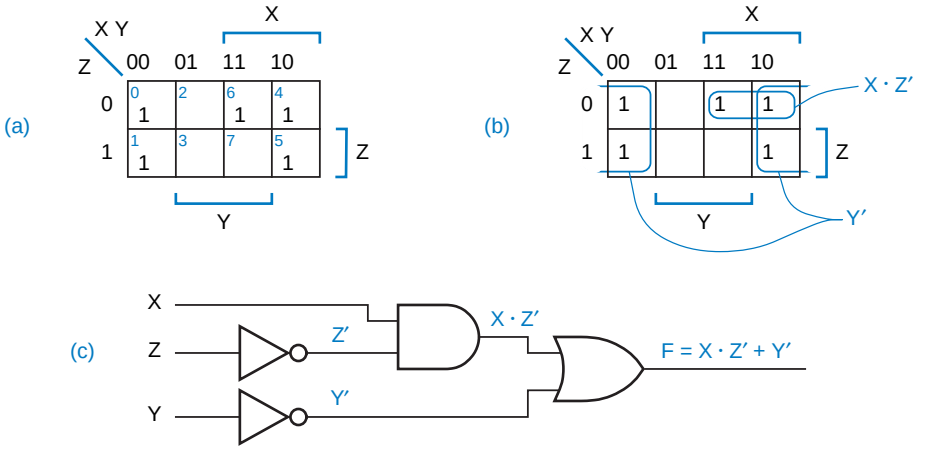


Figure 4-29
$F = \Sigma_{X,Y,Z}(0,1,4,5,6)$: (a) initial Karnaugh map; (b) Karnaugh map with circled product terms; (c) AND/OR circuit.

A sum-of-products expression for a function must contain product terms (circled sets of 1-cells) that cover all of the 1s and none of the 0s on the map.

The Karnaugh map for our most recent example, $F = \Sigma_{X,Y,Z}(0, 1, 4, 5, 6)$, is shown in Figure 4-29(a) and (b). We have circled one set of four 1s, corresponding to the product term Y', and a set of two 1s corresponding to the product term $X \cdot Z'$. Notice that the second product term has one less literal than the corresponding product term in our algebraic solution ($X \cdot Y \cdot Z'$). By circling the largest possible set of 1s containing cell 6, we have found a less expensive realization of the logic function, since a 2-input AND gate should cost less than a 3-input one. The fact that two different product terms now cover the same 1-cell (4) does not affect the logic function, since for logical addition $1 + 1 = 1$, not 2! The corresponding two-level AND/OR circuit is shown in (c).

As another example, the prime-number detector circuit that we introduced in Figure 4-18 on page 215 can be minimized as shown in Figure 4-30.

At this point, we need some more definitions to clarify what we're doing:

minimal sum

- A *minimal sum* of a logic function $F(X_1,\ldots,X_n)$ is a sum-of-products expression for F such that no sum-of-products expression for F has fewer product terms, and any sum-of-products expression with the same number of product terms has at least as many literals.

That is, the minimal sum has the fewest possible product terms (first-level gates and second-level gate inputs) and, within that constraint, the fewest possible literals (first-level gate inputs). Thus, among our three prime-number detector circuits, only the one in Figure 4-30 on the next page realizes a minimal sum.

The next definition says precisely what the word "imply" means when we talk about logic functions:

imply

- A logic function $P(X_1,\ldots,X_n)$ *implies* a logic function $F(X_1,\ldots,X_n)$ if for every input combination such that $P = 1$, then $F = 1$ also.

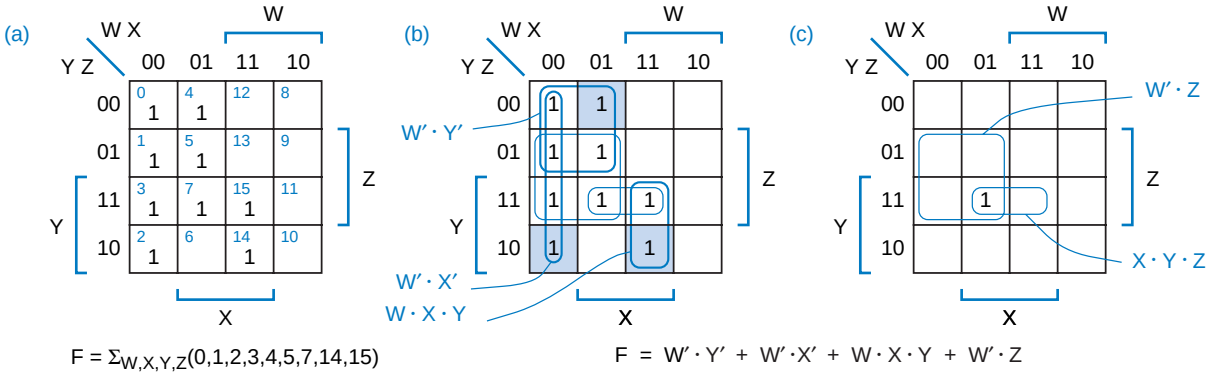


Figure 4-34 $F = \Sigma_{W,X,Y,Z}(0,1,2,3,4,5,7,14,15)$: (a) Karnaugh map; (b) prime implicants and distinguished 1-cells; (c) reduced map after removal of essential prime implicants and covered 1-cells.

An example of eclipsing is shown in Figure 4-35. After removing essential prime implicants, we are left with two 1-cells, each of which is covered by two prime implicants. However, $X \cdot Y \cdot Z$ eclipses the other two prime implicants, which therefore may be removed from consideration. The two 1-cells are then covered only by $X \cdot Y \cdot Z$, which is a *secondary essential prime implicant* that must be included in the minimal sum.

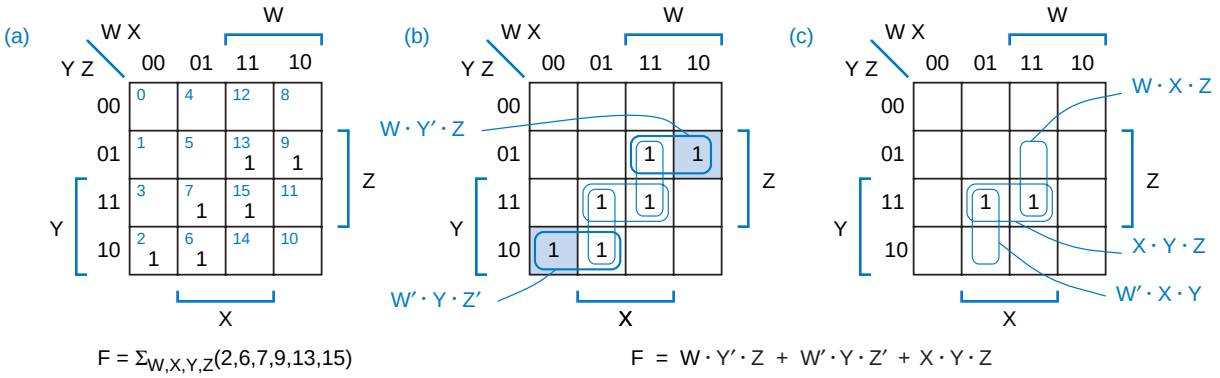
secondary essential prime implicant

Figure 4-36 shows a more difficult case—a logic function with no essential prime implicants. By trial and error we can find two different minimal sums for this function.

We can also approach the problem systematically using the *branching method*. Starting with any 1-cell, we arbitrarily select one of the prime implicants that covers it, and we include it as if it were essential. This simplifies the

branching method

Figure 4-35 $F = \Sigma_{W,X,Y,Z}(2,6,7,9,13,15)$: (a) Karnaugh map; (b) prime implicants and distinguished 1-cells; (c) reduced map after removal of essential prime implicants and covered 1-cells.



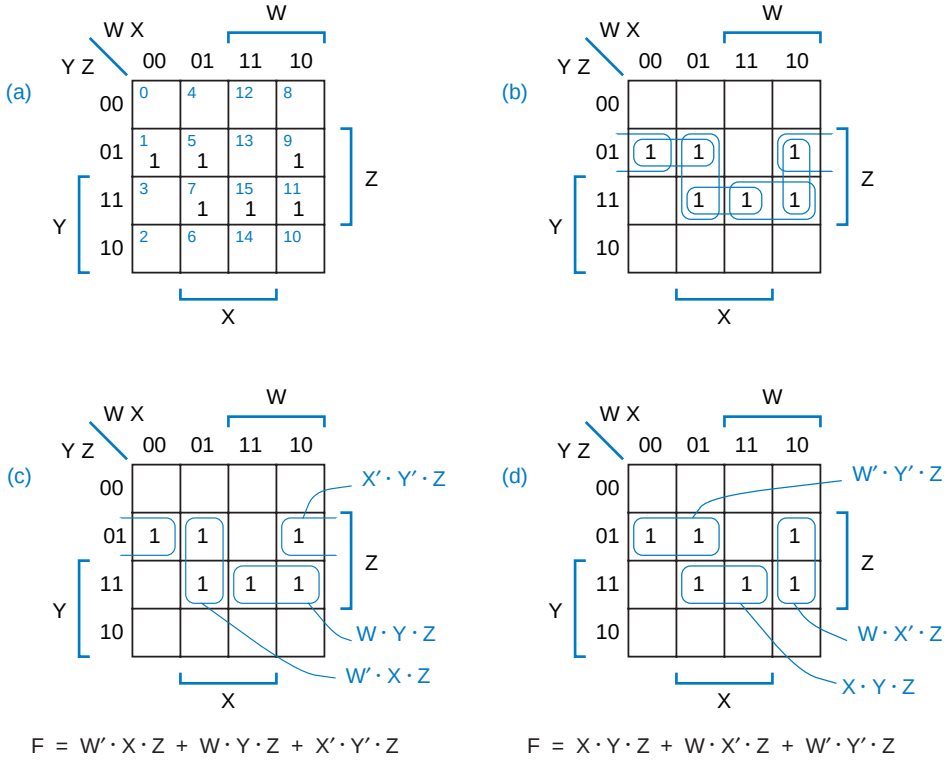


Figure 4-36 $F = \Sigma_{W,X,Y,Z}(1,5,7,9,11,15)$: (a) Karnaugh map; (b) prime implicants; (c) a minimal sum; (d) another minimal sum.

remaining problem, which we can complete in the usual way to find a tentative minimal sum. We repeat this process starting with all other prime implicants that cover the starting 1-cell, generating a different tentative minimal sum from each starting point. We may get stuck along the way and have to apply the branching method recursively. Finally, we examine all of the tentative minimal sums that we generated in this way and select one that is truly minimal.

4.3.6 Simplifying Products of Sums

Using the principle of duality, we can minimize product-of-sums expressions by looking at the 0s on a Karnaugh map. Each 0 on the map corresponds to a maxterm in the canonical product of the logic function. The entire process in the preceding subsection can be reformulated in a dual way, including the rules for writing sum terms corresponding to circled sets of 0s, in order to find a *minimal product*.

minimal product

Fortunately, once we know how to find minimal sums, there's an easier way to find the minimal product for a given logic function F. The first step is to complement F to obtain F′. Assuming that F is expressed as a minterm list or a

The remainder of the procedure is the same. In particular, we look for distinguished 1-cells and *not* distinguished d-cells, and we include only the corresponding essential prime implicants and any others that are needed to cover all the 1s on the map. In Figure 4-37, the two essential prime implicants are sufficient to cover all of the 1s on the map. Two of the d's also happen to be covered, so F will be 1 for don't-care input combinations 10 and 11, and 0 for the other don't-cares.

Some HDLs, including ABEL, provide a means for the designer to specify don't-care inputs, and the logic-minimization program takes these into account when computing a minimal sum.

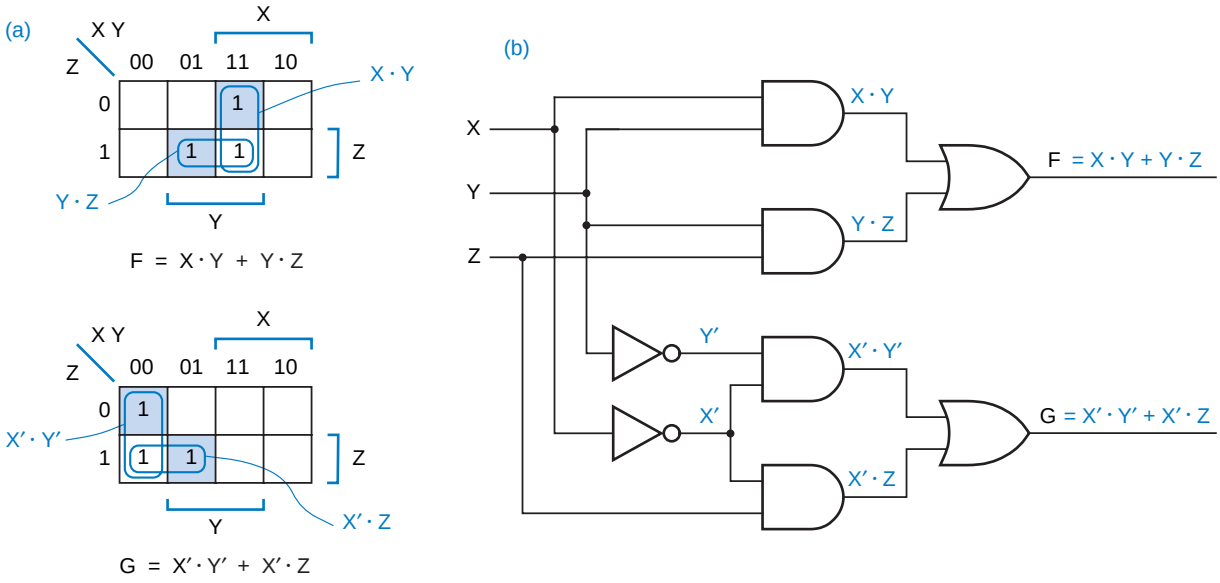
*4.3.8 Multiple-Output Minimization

Most practical combinational logic circuits require more than one output. We can always handle a circuit with *n* outputs as *n* independent single-output design problems. However, in doing so, we may miss some opportunities for optimization. For example, consider the following two logic functions:

$$F = \Sigma_{X,Y,Z}(3,6,7)$$
$$G = \Sigma_{X,Y,Z}(0,1,3)$$

Figure 4-38 shows the design of F and G as two independent single-output functions. However, as shown in Figure 4-39, we can also find a pair of sum-of-products expressions that share a product term, such that the resulting circuit has one fewer gate than our original design.

Figure 4-38 Treating a 2-output design as two independent single-output designs: (a) Karnaugh maps; (b) "minimal" circuit.



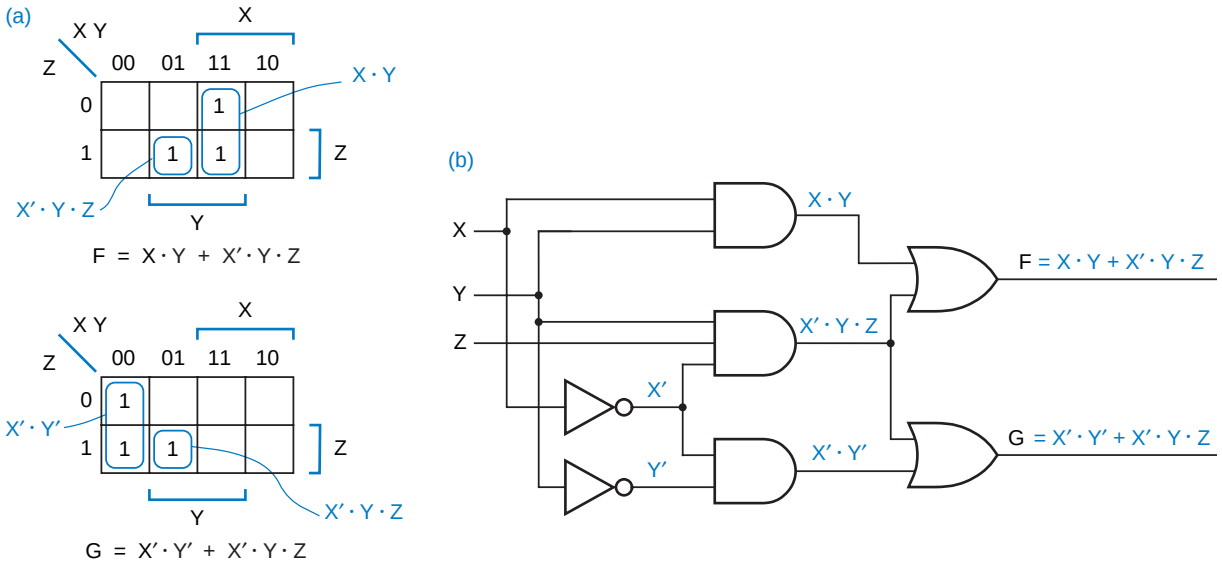


Figure 4-39 Multiple-output minimization for a 2-output circuit: (a) minimized maps including a shared term; (b) minimal multiple-output circuit

When we design multiple-output combinational circuits using discrete gates, as in an ASIC, product-term sharing obviously reduces circuit size and cost. In addition, PLDs contain multiple copies of the sum-of-products structure that we've been learning how to minimize, one per output, and some PLDs allow product terms to be shared among multiple outputs. Thus, the ideas introduced in this subsection are used in many logic-minimization programs.

You probably could have "eyeballed" the Karnaugh maps for F and G in Figure 4-39 and discovered the minimal solution. However, larger circuits can be minimized only with a formal multiple-output minimization algorithm. We'll outline the ideas in such an algorithm here; details can be found in the References.

m-product function

The key to successful multiple-output minimization of a set of n functions is to consider not only the n original single-output functions, but also "product functions." An *m-product function* of a set of n functions is the product of m of the functions, where $2 \le m \le n$. There are $2^n - n - 1$ such functions. Fortunately, $n = 2$ in our example and there is only one product function, F · G, to consider. The Karnaugh maps for F, G, and F · G are shown in Figure 4-40; in general, the map for an m-product function is obtained by ANDing the maps of its m components.

multiple-output prime implicant

A *multiple-output prime implicant* of a set of n functions is a prime implicant of one of the n functions or of one of the product functions. The

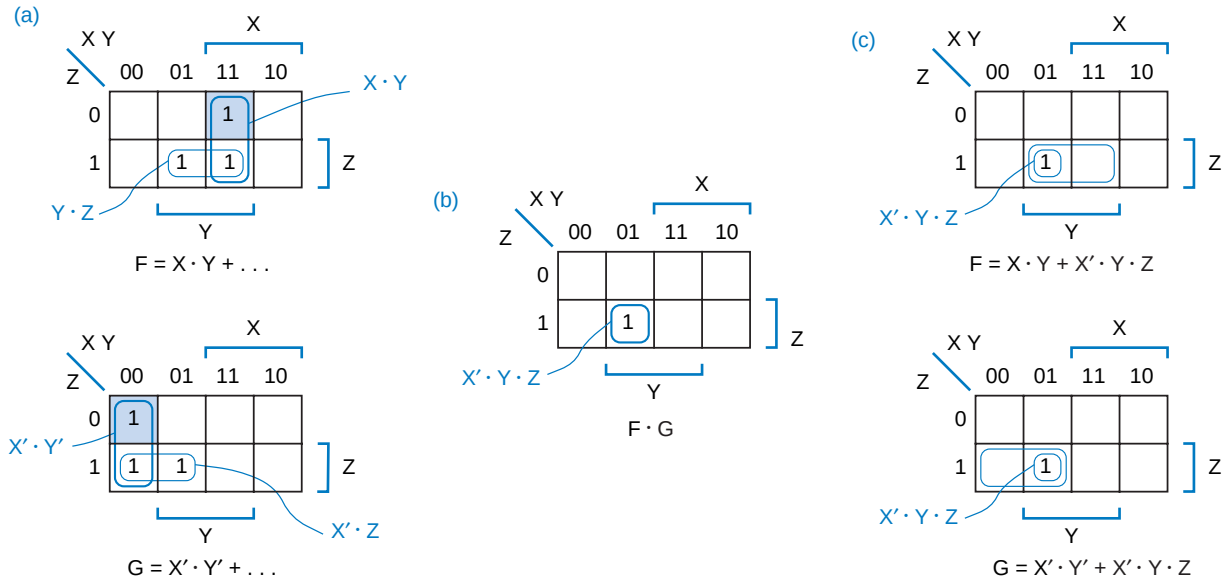


Figure 4-40 Karnaugh maps for a set of two functions: (a) maps for F and G; (b) 2-product map for F · G; (c) reduced maps for F and G after removal of essential prime implicants and covered 1-cells.

first step in multiple-output minimization is to find all of the multiple-output prime implicants. Each prime implicant of an *m*-product function is a possible term to include in the corresponding *m* outputs of the circuit. If we were trying to minimize a set of 8 functions, we would have to find the prime implicants for $2^8 - 8 - 1 = 247$ product functions as well as for the 8 given functions. Obviously, multiple-output minimization is not for the faint-hearted!

Once we have found the multiple-output prime implicants, we try to simplify the problem by identifying the essential ones. A *distinguished 1-cell* of a particular single-output function F is a 1-cell that is covered by exactly one prime implicant of F or of the product functions involving F. The distinguished 1-cells in Figure 4-40 are shaded. An *essential prime implicant* of a particular single-output function is one that contains a distinguished 1-cell. As in single-output minimization, the essential prime implicants must be included in a minimum-cost solution. Only the 1-cells that are not covered by essential prime implicants are considered in the remainder of the algorithm.

distinguished 1-cell

essential prime implicant

The final step is to select a minimal set of prime implicants to cover the remaining 1-cells. In this step we must consider all *n* functions simultaneously, including the possibility of sharing; details of this procedure are discussed in the References. In the example of Figure 4-40(c), we see that there exists a single, shared product term that covers the remaining 1-cell in both F and G.

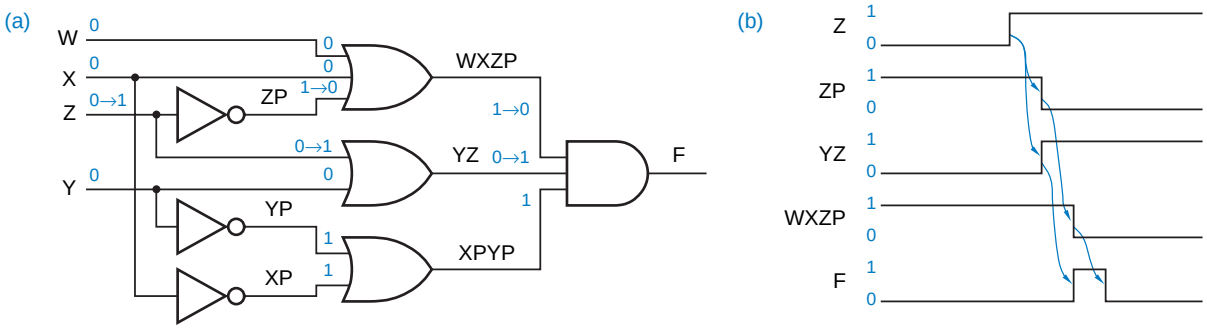


Figure 4-45 Circuit with static-0 hazards: (a) logic diagram; (b) timing diagram.

variable and its complement were connected to the same AND gate, which would be silly. However, the circuit *may* have static-1 hazards. Their existence can be predicted from a Karnaugh map where the product terms corresponding to the AND gates in the circuit are circled.

Figure 4-46(a) shows the Karnaugh map for the circuit of Figure 4-44. It is clear from the map that there is no single product term that covers both input combinations X,Y,Z = 111 and X,Y,Z = 110. Thus, intuitively, it is possible for the output to "glitch" momentarily to 0 if the AND gate output that covers one of the combinations goes to 0 before the AND gate output covering the other input combination goes to 1. The way to eliminate the hazard is also quite apparent: Simply include an extra product term (AND gate) to cover the hazardous input pair, as shown in Figure 4-46(b). The extra product term, it turns out, is the *consensus* of the two original terms; in general, we must add consensus terms to eliminate hazards. The corresponding hazard-free circuit is shown in Figure 4-47.

consensus

Another example is shown in Figure 4-48. In this example, three product terms must be added to eliminate the static-1 hazards.

A properly designed two-level product-of-sums (OR-AND) circuit has no static-1 hazards. It *may* have static-0 hazards, however. These hazards can be detected and eliminated by studying the adjacent 0s in the Karnaugh map, in a manner dual to the foregoing.

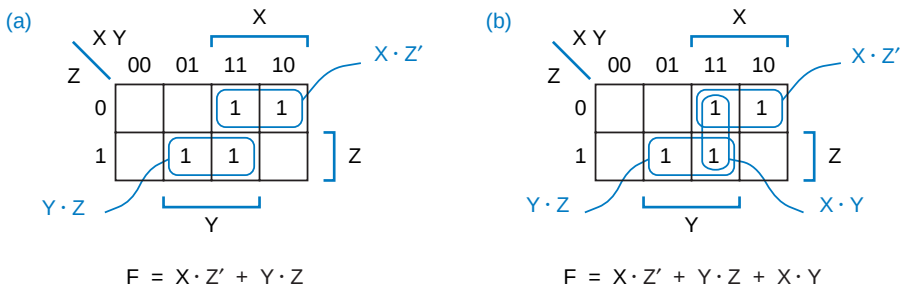


Figure 4-46
Karnaugh map for the circuit of Figure 4-44: (a) as originally designed; (b) with static-1 hazard eliminated.

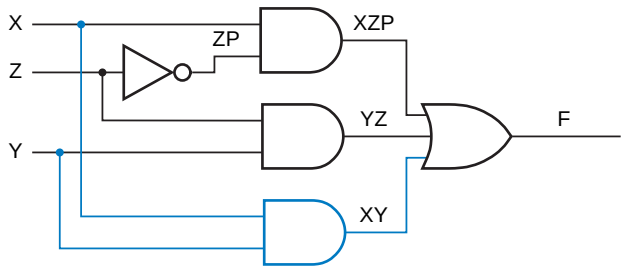


Figure 4-47
Circuit with static-1 hazard eliminated.

*4.5.3 Dynamic Hazards

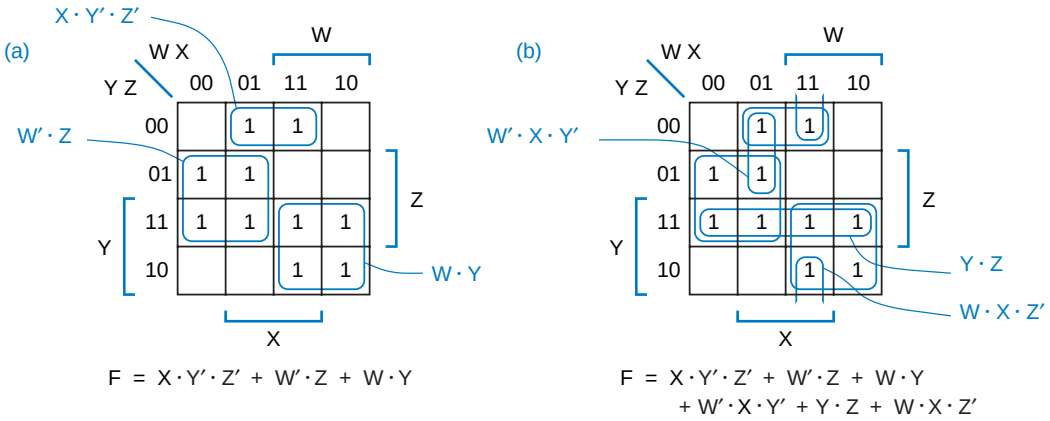
dynamic hazard

A *dynamic hazard* is the possibility of an output changing more than once as the result of a single input transition. Multiple output transitions can occur if there are multiple paths with different delays from the changing input to the changing output.

For example, consider the circuit in Figure 4-49; it has three different paths from input X to the output F. One of the paths goes through a slow OR gate, and another goes through an OR gate that is even slower. If the input to the circuit is $W, X, Y, Z = 0, 0, 0, 1$, then the output will be 1, as shown. Now suppose we change the X input to 1. Assuming that all of the gates except the two marked "slow" and "slower" are very fast, the transitions shown in black occur next, and the output goes to 0. Eventually, the output of the "slow" OR gate changes, creating the transitions shown in nonitalic color, and the output goes to 1. Finally, the output of the "slower" OR gate changes, creating the transitions shown in italic color, and the output goes to its final state of 0.

Dynamic hazards do not occur in a properly designed two-level AND-OR or OR-AND circuit, that is, one in which no variable and its complement are con-

Figure 4-48 Karnaugh map for another sum-of-products circuit: (a) as originally designed; (b) with extra product terms to cover static-1 hazards.



```
test_vectors
([PANIC,ENABLEA,EXITING,WINDOW,DOOR,GARAGE] -> [ALARM])
[     1,      X,      X,      X,    X,      X] -> [    1];  "1
[     0,      0,      X,      X,    X,      X] -> [    0];  "2
[     0,      1,      1,      X,    X,      X] -> [    0];  "3
[     0,      1,      0,      0,    X,      X] -> [    1];  "4
[     0,      1,      0,      X,    0,      X] -> [    1];  "5
[     0,      1,      0,      X,    X,      0] -> [    1];  "6
[     0,      1,      0,      1,    1,      1] -> [    0];  "7
```

Table 4-24 Test vectors for the alarm circuit program in Table 4-11.

The problem is that ABEL doesn't handle don't-cares in test-vector inputs the way that it should. For example, by all rights, test vector 1 should test 32 distinct input combinations corresponding to all 32 possible combinations of don't-care inputs ENABLEA, EXITING, WINDOW, DOOR, and GARAGE. But it doesn't. In this situation, the ABEL compiler interprets "don't care" as "the user doesn't care what input value I use," and it just assigns 0 to all don't-care inputs in a test vector. In this example, you could have incorrectly written the output equation as "F = PANIC & !ENABLEA # ENABLEA & . . ."; the test vectors would still pass, even though the panic button would work only when the system is disabled.

The second use of test vectors is in physical device testing. Most physical defects in logic devices can be detected using the *single stuck-at fault model*, which assumes that any physical defect is equivalent to having a single gate input or output stuck at a logic 0 or 1 value. Just putting together a set of test vectors that seems to exercise a circuit's functional specifications, as we did in Table 4-24, doesn't guarantee that all single stuck-at faults can be detected. The test vectors have to be chosen so that every possible stuck-at fault causes an incorrect value at the circuit output for some test-vector input combination.

single stuck-at fault model

Table 4-25 shows a complete set of test vectors for the alarm circuit when it is realized as a two-level sum-of-products circuit. The first four vectors check for stuck-at-1 faults on the OR gate, and the last three check for stuck-at-0 faults on the AND gates; it turns out that this is sufficient to detect all single stuck-at faults. If you know something about fault testing you can generate test vectors for small circuits by hand (as I did in this example), but most designers use automated third-party tools to create high-quality test vectors for their PLD designs.

```
test_vectors
([PANIC,ENABLEA,EXITING,WINDOW,DOOR,GARAGE] -> [ALARM])
[     1,      0,      1,      1,    1,      1] -> [    1];  "1
[     0,      1,      0,      0,    1,      1] -> [    1];  "2
[     0,      1,      0,      1,    0,      1] -> [    1];  "3
[     0,      1,      0,      1,    1,      0] -> [    1];  "4
[     0,      0,      0,      0,    0,      0] -> [    0];  "5
[     0,      1,      1,      0,    0,      0] -> [    0];  "6
[     0,      1,      0,      1,    1,      1] -> [    0];  "7
```

Table 4-25 Single stuck-at fault test vectors for the minimal sum-of-products realization of the alarm circuit.