ECE 2372 Modern Digital Design July 13

Reading assignment for today: Ch 2: section 2-4

Let us review the last problem we looked at on Friday. In that example on Friday, we actually left out one of the coverings of a pair of 1's in the K-map. It turns out they were already covered by other coverings, so the solution we worked out was valid. The additional covering, had we included it would have simply been redundant.

So, let us set up the problem again, but this time let's include that additional redundant covering in the map.

It is also true that moving from one row to the next results in only one bit position change. Moving from any one cell to an adjacent cell only involves one bit position change.

Now let's map a function into this K-map

Binary code			Z= F(A,B,C)	
A	B	C		
0	0	0	0	m0
0	0	1	0	m1
0	1	0	1	m2
0	1	1	1	m3
1	0	0	1	m4
1	0	1	1	m5
1	1	0	1	m6
1	1	1	0	m7

I have just made up an arbitrary function shown in the truth table above. We can also write it out in other forms.

Z = F(A,B,C) = ~AB~C + ~ABC + A~B~C + A~BC + AB~C

Or, the more compact format:

Z = F(A,B,C) = Σm (2,3,4,5,6)

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

Note we get four groupings this time. We have some additional overlap. That is OK. It is acceptable to "cover" a given cell position in more than one way, but all cells with 1's in them have to be included in the final solution, even if it is just a single cell. If two cells are covered in a 3-variable map, then only two variables or terms are necessary to define that cell grouping. For example, let's look at the upper right hand corner grouping of the minterm cells 011 (~ABC) and 010 (~AB~C). This grouping is in the row where A = 0, so A has to be included. Notice that this same grouping overlaps the case of C = 0 and C = 1, so the variable C is not relevant here since the grouping covers both cases for C. However, note this grouping covers the case where B = 1 but not where B = 0. So, this grouping covers the term X = 0 and B = 1. We have two other groupings. Bottom left corner is similar, but covers the case where A = 1 and B = 0, but for either case of C. Then the last grouping, the vertical rectangle that groups together the case where C = 0, B = 1, and A = either case.

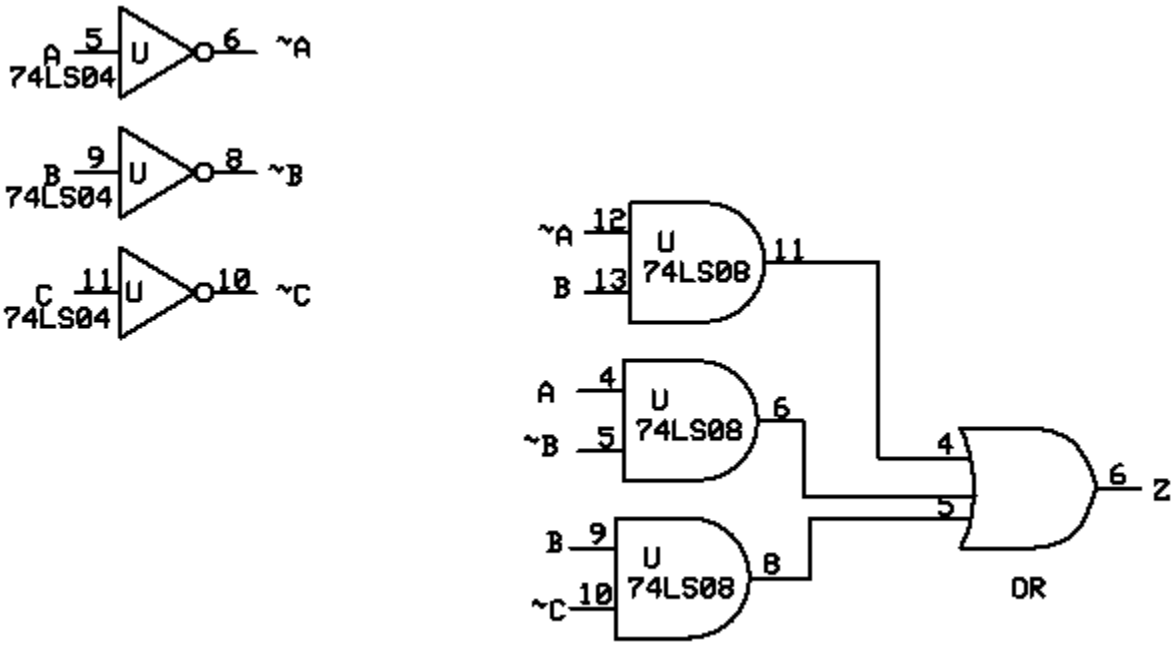
A 1 in the K-map only has to be covered once. It should included in the largest grouping if it is in more than one grouping. Additional coverings beyond are redundant and make the solution not a minimal solution.

Note that A~B~C and A~BC are covered. A~B~C can be grouped with AB~C to form a group of two, but it results in a redundant covering and is not needed in this case.

So we wind up with 3 groupings of two each:

~AB + A~B + B~C

This is now in Sum of Products (SOP) form, and the SOP implementation with logic gates is given below:



Sometimes we have conditions in a logic design that are “don’t care” situations. Let us review the above problem but include some “don’t care” terms and see how that changes things. We can show this in a truth table as shown:

Now let’s map a function into this K-map

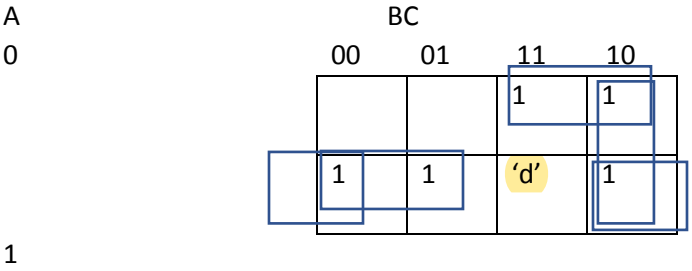
Binary code A	B	C	Z= F(A,B,C)	
0	0	0	0	m0
0	0	1	0	m1
0	1	0	1	m2
0	1	1	1	m3
1	0	0	1	m4
1	0	1	1	m5
1	1	0	1	m6
1	1	1	d	m7

I have just made up an arbitrary function shown in the truth table above. We can also write it out in other forms.

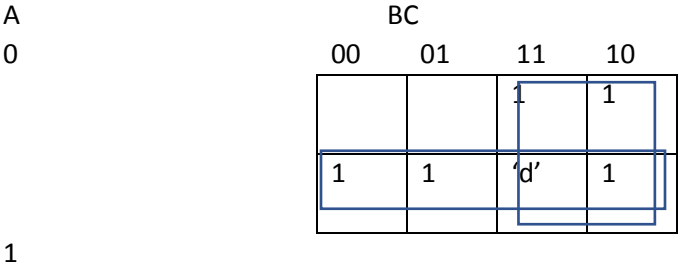
Z = F(A,B,C) = ~AB~C + ~ABC + A~B~C + A~BC + AB~C , don’t care ABC

Or, the more compact format:

Z = F(A,B,C) = Σm (2,3,4,5,6, d7)



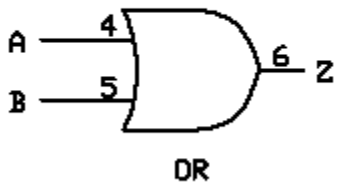
Notice we have placed a ‘d’ for don’t care in the cell ABC, minterm 7. Because of that, we can now regroup the problem in a more minimal way.



Notice we now have only two groupings. We have A and we have B. The function has reduced to:

Z = F(A,B,C) = A + B

The logic circuit reduces to:



Although this does not appear to be a two level SOP solution, it is a simple variation of that. Don't cares can often considerably simplify a logic design.

We have discussed so far the SOP (Sum of Products) minimization approach using minterms. In the SZOP format, AND gates are used on the input or first level to form the product term and OR gates are used on the second or output level to form the SUM term.

For POS (Product of Sums), we instead use Maxterms (where the function is ZEZRO) and use OR gates on the input level and AND gates on the output level.

A page from your text book showing a table of Maxterms is given below:

Notice the table starts off with X + Y + Z for the first Maxterm and then goes thru ~X +~Y + ~Z for the last Maxterm. Also, note that Maxterms use the case where the function is ZERO, not where it is ONE.

Now let's map a function into this K-map

Binary code			Z= F(A,B,C)	
A	B	C		
0	0	0	0	M0
0	0	1	0	M1
0	1	0	1	M2
0	1	1	1	M3
1	0	0	1	M4
1	0	1	1	M5
1	1	0	1	M6
1	1	1	d	M7

The function is ZERO for Maxterms 0 and 1

F = Mπ(0,1) = 0

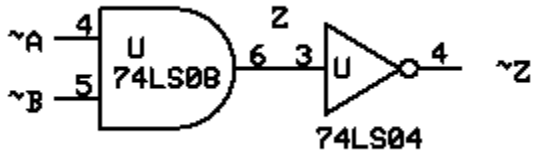
A	BC			
0	00	01	11	10
			1	1
1	1	1	'd'	1

To solve this K-Map of Maxterms, we circle groups of 0's. Groups of powers of two of ZEZROS, so one ZERO, two ZEROS, four ZEROS.

This reduces to F M(A,B,C) = 0 = ~A~B

Notice the "don't care" did not help us here.

The final POS solution for this simple problem only has one layer, the AND output layer. The availability of the inverse of signals A and B is assumed. In any case, they are easily obtained with inverter gates.



I will post a practice quiz tomorrow. We will have a real quiz on Wednesday and our first exam on Thursday. I will post a solution for the practice quiz, and I will post a solution for the actual quiz after it is complete. The actual exam will cover things similar to the homework problems and will involve binary and hexadecimal numbers, some simple Boolean Algebra, truth tables, and simple SOP K-map reductions. I will also be posting solutions for the homeworks in the next day or so. Turn in the homeworks according to the instructions provide on BB. A third homework over K-map reductions will be posted on BB today.

and Table 2-10 that a minterm and maxterm with the same subscript are the complements of each other; that is, $M_j = \overline{m}_j$ and $m_j = \overline{M}_j$. For example, for $j = 3$, we have

$$M_3 = X + \overline{Y} + \overline{Z} + = \overline{\overline{X}YZ} = \overline{m}_3$$

A Boolean function can be represented algebraically from a given truth table by forming the logical sum of all the minterms that produce a 1 in the function. This expression is called a *sum of minterms*. Consider the Boolean function F in Table 2-11(a). The function is equal to 1 for each of the following binary combinations of the variables X, Y, and Z: 000, 010, 101 and 111. These combinations correspond to minterms 0, 2, 5, and 7. By examining Table 2-11 and the truth tables for these minterms in Table 2-9, it is evident that the function F can be expressed algebraically as the logical sum of the stated minterms:

$$F = \overline{X}\,\overline{Y}\,\overline{Z} + \overline{X}Y\overline{Z} + X\overline{Y}Z + XYZ = m_0 + m_2 + m_5 + m_7$$

This can be further abbreviated by listing only the decimal subscripts of the minterms:

$$F(X, Y, Z) = \Sigma m(0, 2, 5, 7)$$

□ **TABLE 2-10**
Maxterms for Three Variables

X	Y	Z	Sum Term	Symbol	M_0	M_1	M_2	M_3	M_4	M_5	M_6	M_7
0	0	0	$X + Y + Z$	M_0	0	1	1	1	1	1	1	1
0	0	1	$X + Y + \overline{Z}$	M_1	1	0	1	1	1	1	1	1
0	1	0	$X + \overline{Y} + Z$	M_2	1	1	0	1	1	1	1	1
0	1	1	$X + \overline{Y} + \overline{Z}$	M_3	1	1	1	0	1	1	1	1
1	0	0	$\overline{X} + Y + Z$	M_4	1	1	1	1	0	1	1	1
1	0	1	$\overline{X} + Y + \overline{Z}$	M_5	1	1	1	1	1	0	1	1
1	1	0	$\overline{X} + \overline{Y} + Z$	M_6	1	1	1	1	1	1	0	1
1	1	1	$\overline{X} + \overline{Y} + \overline{Z}$	M_7	1	1	1	1	1	1	1	0

□ **TABLE 2-11**
Boolean Functions of Three Variables

(a) X	Y	Z	F	$\overline{F}$	(b) X	Y	Z	E
0	0	0	1	0	0	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	1	0	0	1	0	1
0	1	1	0	1	0	1	1	0
1	0	0	0	1	1	0	0	1
1	0	1	1	0	1	0	1	1
1	1	0	0	1	1	1	0	0
1	1	1	1	0	1	1	1	0

4.3 Combinational-Circuit Synthesis

What is the starting point for designing combinational logic circuits? Usually, we are given a word description of a problem or we develop one ourselves. In modern digital-design environments, we can translate that word description into a program in a hardware description language (HDL), and the HDL tools can synthesize a circuit structure for that program, targeting a selected device or technology (such as an FPGA or an ASIC cell library). In such a case, the designer never gets involved in synthesis at all. However, even in modern environments, there are still many situations where the designer should be capable of synthesizing combinational circuits "by hand," and this section introduces the basic skills needed to perform such synthesis.

4.3.1 Circuit Descriptions and Designs

Occasionally, a logic circuit description is just a list of input combinations for which a signal should be on or off, the verbal equivalent of a truth table or the Σ or Π notation introduced previously. For example, the description of a 4-bit prime-number detector might be, "Given a 4-bit input combination $N = N_3N_2N_1N_0$, produce a 1 output for $N = 1, 2, 3, 5, 7, 11, 13$, and 0 otherwise." A logic function described in this way can be designed directly from the canonical sum or product expression. For the prime-number detector, we have

$$\begin{aligned} F &= \Sigma_{N_3,N_2,N_1,N_0}(1, 2, 3, 5, 7, 11, 13) \\ &= N_3' \cdot N_2' \cdot N_1' \cdot N_0 + N_3' \cdot N_2' \cdot N_1 \cdot N_0' + N_3' \cdot N_2' \cdot N_1 \cdot N_0 + N_3' \cdot N_2 \cdot N_1' \cdot N_0 \\ &\quad + N_3' \cdot N_2 \cdot N_1 \cdot N_0 + N_3 \cdot N_2' \cdot N_1 \cdot N_0 + N_3 \cdot N_2 \cdot N_1' \cdot N_0 \end{aligned}$$

ROLLING YOUR OWN

There are sometimes cases when an HDL or an appropriate synthesizer i able, or the synthesizer's results just aren't good enough. The last case i common when a designer needs to optimize the speed, power consumpt of a critical circuit. For example, in a microprocessor system-on-a-chip w of gates, an HDL-based approach is essential for the "routine" parts of t achieve a reasonable time to market. Yet, it is still very common for paths, such as adders, multipliers, multiplexers, and specialized high-sp structures to be synthesized "by hand," with no synthesis tool, and the used at most to specify the desired interconnection between individua larger hand-designed logic modules (e.g., adders, multiplexers, and deco

There are also cases where the synthesizer may "run amok," creati that is much less efficient (in speed, size, or some other metric) than what and required. In these cases, it is important for the designer to have a g what *could* be achieved, and either synthesize the circuit by hand or try style of HDL coding to try to cajole the synthesizer into creating a result t to what is desired.

PRIME TIME Mathematicians will tell you that "1" is not really a prime number. But our prime-number detector example is not nearly as interesting, from a logic-design point of view, if "1" is not prime. So, please go do Drill 4.11 if you want to be a mathematical purist.

The corresponding circuit is shown in Figure 4-18.

More often, we describe a logic function using the English-language connectives "and," "or," and "not." For example, we might describe an alarm circuit by saying, "The ALARM output is 1 if the PANIC input is 1, or if the ENABLE input is 1, the EXITING input is 0, and the house is not secure; the house is secure if the WINDOW, DOOR, and GARAGE inputs are all 1." Such a description can be translated directly into algebraic expressions:

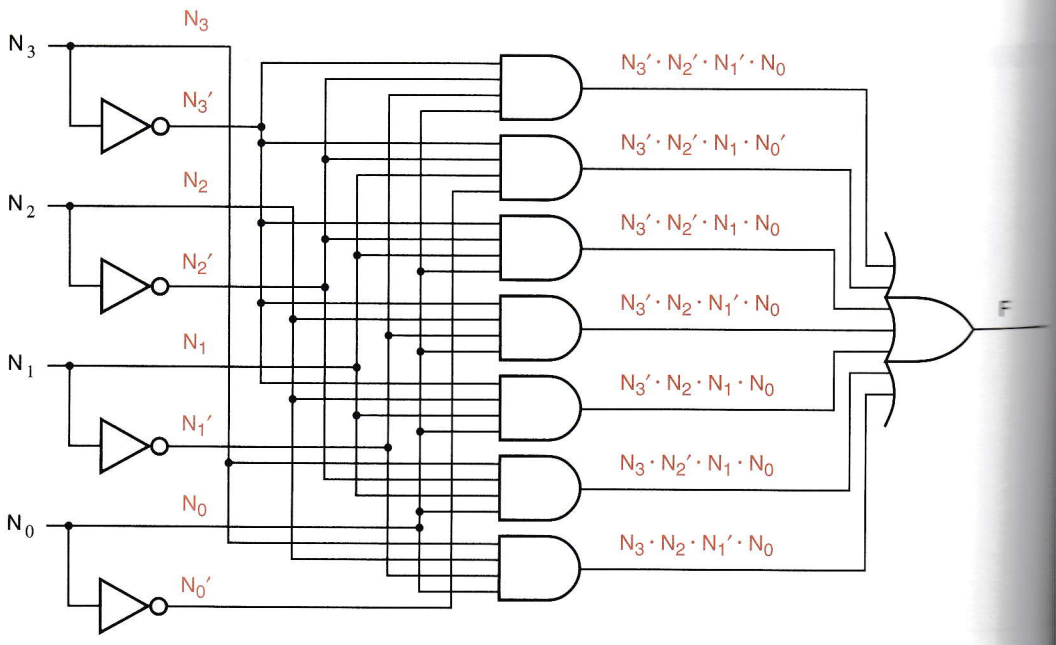
$$\text{ALARM} = \text{PANIC} + \text{ENABLE} \cdot \text{EXITING}' \cdot \text{SECURE}'$$

$$\text{SECURE} = \text{WINDOW} \cdot \text{DOOR} \cdot \text{GARAGE}$$

$$\text{ALARM} = \text{PANIC} + \text{ENABLE} \cdot \text{EXITING}' \cdot (\text{WINDOW} \cdot \text{DOOR} \cdot \text{GARAGE})$$

Notice that we used the same method in switching algebra as in ordinary algebra to formulate a complicated expression—we defined an auxiliary variable SECURE to simplify the first equation, developed an expression for SECURE, and used substitution to get the final expression. We can easily draw a circuit

Figure 4-18 Canonical-sum design for 4-bit prime-number detector.



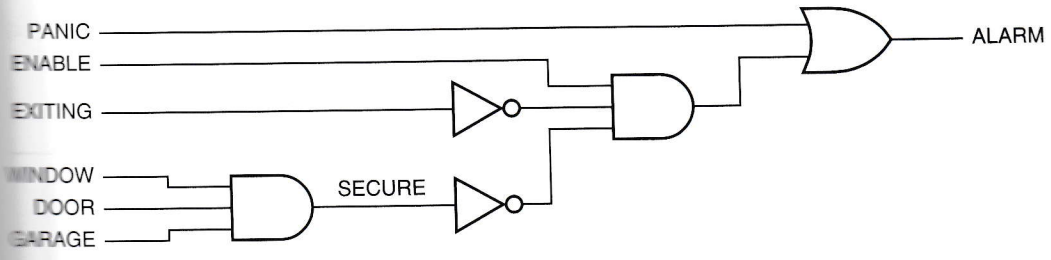


Figure 4-19 Alarm circuit derived from logic expression.

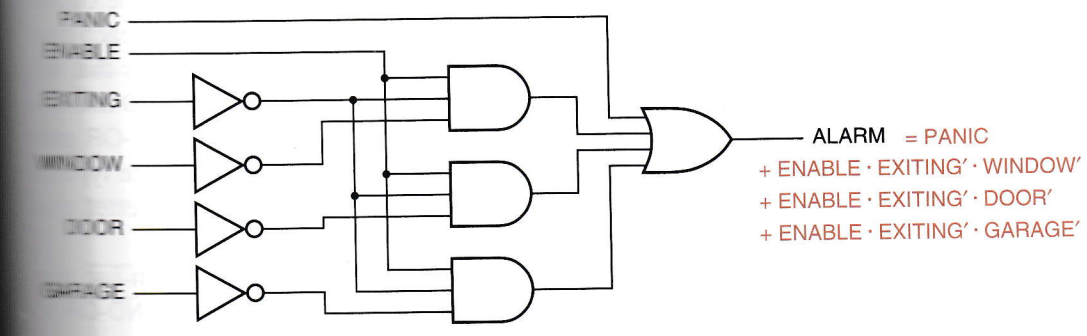
using AND, OR, and NOT gates that realizes the final expression, as shown in Figure 4-19. A circuit *realizes* ["makes real"] an expression if its output function equals that expression, and the circuit is called a *realization* of the function.

realize
realization

Once we have an expression, any expression, for a logic function, we can do other things besides building a circuit directly from the expression. We can manipulate the expression to get different circuits. For example, the ALARM expression above can be multiplied out to get the sum-of-products circuit in Figure 4-20. Or, if the number of variables is not too large, we can construct the truth table for the expression and use any of the synthesis methods that apply to truth tables, including the canonical sum or product method described earlier and the minimization methods described later.

In general, it's easier to describe a circuit in words using logical connectives and to write the corresponding logic expressions than it is to write a complete truth table, especially if the number of variables is large. However, sometimes we have to work with imprecise word descriptions of logic functions, for example, "The ERROR output should be 1 if the GEARUP, GEARDOWN, and GEARCHECK inputs are inconsistent." In this situation, the truth-table approach is best because it allows us to determine the output required for every input combination, based on our knowledge and understanding of the problem environment (e.g., the brakes cannot be applied unless the gear is down).

Figure 4-20 Sum-of-products version of alarm circuit.



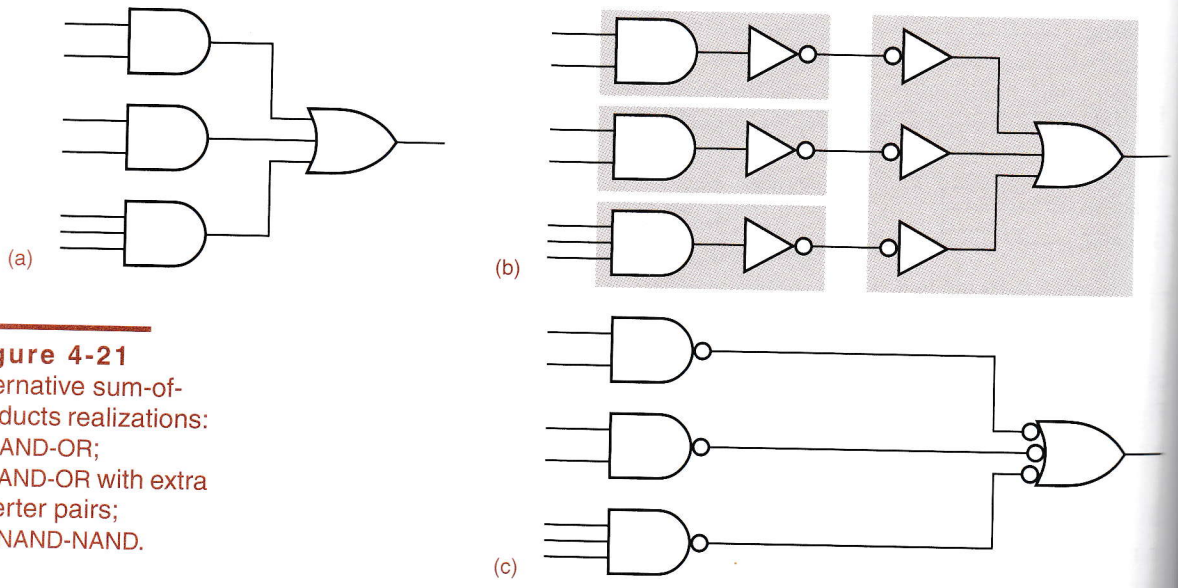


Figure 4-21
Alternative sum-of-products realizations: (a) AND-OR; (b) AND-OR with extra inverter pairs; (c) NAND-NAND.

4.3.2 Circuit Manipulations

The design methods that we've described so far use AND, OR, and NOT gates. We might like to use NAND and NOR gates, too—they're faster than ANDs and ORs in most technologies. However, most people don't develop logical propositions in terms of NAND and NOR connectives. That is, you probably wouldn't say, "I won't date you if you're not clean or not wealthy and also you're not smart or not friendly." It would be more natural for you to say, "I'll date you if you're clean and wealthy, or if you're smart and friendly." So, given a "natural" logic expression, we need ways to translate it into other forms.

We can translate any logic expression into an equivalent sum-of-products expression, simply by multiplying it out. As shown in Figure 4-21(a), such an expression may be realized directly with AND and OR gates. The inverters required for complemented inputs are not shown.

As shown in Figure 4-21(b), we may insert a pair of inverters between each AND-gate output and the corresponding OR-gate input in a two-level AND-OR circuit. According to theorem T4, these inverters have no effect on the output function of the circuit. In fact, we've drawn the second inverter of each pair with its inversion bubble on its input to provide a graphical reminder that the inverters cancel. However, if these inverters are absorbed into the AND and OR gates, we wind up with AND-NOT gates at the first level and a NOT-OR gate at the second level. These are just two different symbols for the same type of gate—a NAND gate. Thus, a two-level *AND-OR circuit* may be converted to a two-level *NAND-NAND circuit* simply by substituting gates.

AND-OR circuit
NAND-NAND circuit

If any product terms in the sum-of-products expression contain just one literal, then we may gain or lose inverters in the transformation from AND-OR to

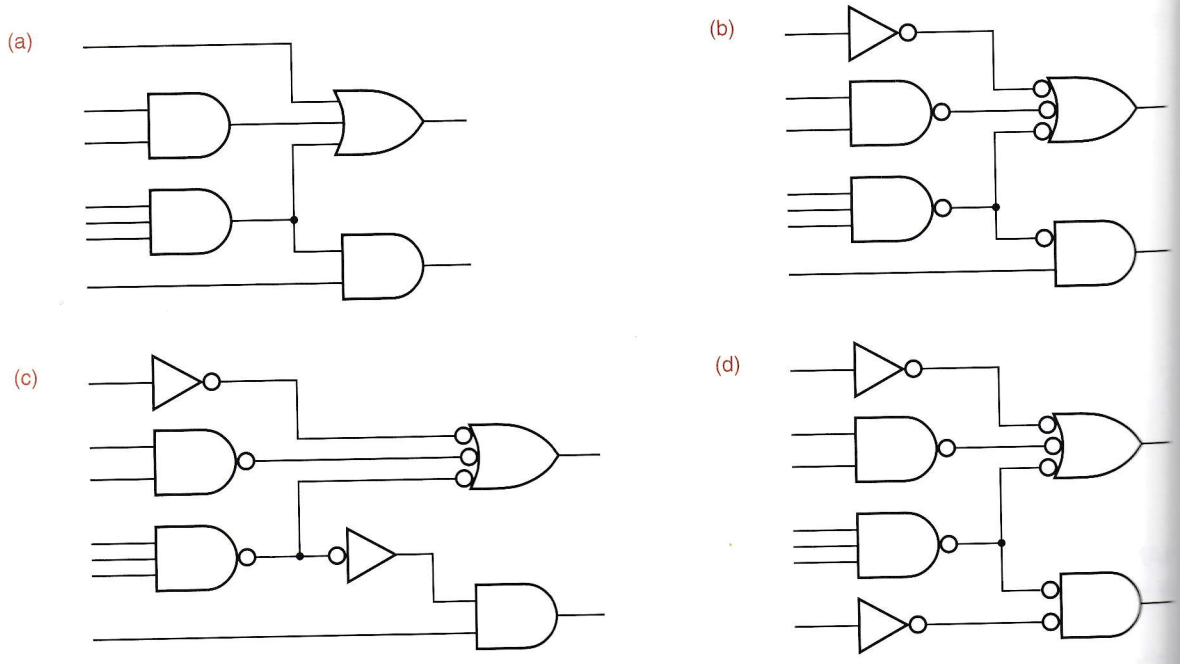


Figure 4-24 Logic-symbol manipulations: (a) original circuit; (b) transformation with a nonstandard gate; (c) inverter used to eliminate nonstandard gate; (d) preferred inverter placement.

The same kind of manipulations can be applied to arbitrary logic circuits. For example, Figure 4-24(a) shows a circuit built from AND and OR gates. After adding pairs of inverters, we obtain the circuit in (b). However, one of the gates, a 2-input AND gate with a single inverted input, is not a standard type. We can use a discrete inverter as shown in (c) to obtain a circuit that uses only standard gate types—NAND, AND, and inverters. Actually, a better way to use the inverter is shown in (d); one level of gate delay is eliminated, and the bottom gate becomes a NOR instead of AND. In most logic technologies, inverting gates like NAND and NOR are faster than noninverting gates like AND and OR.

4.3.3 Combinational-Circuit Minimization

It's often uneconomical to realize a logic circuit directly from the first logic expression that pops into your head. Canonical sum and product expressions are especially expensive because the number of possible minterms or maxterms (and hence gates) grows exponentially with the number of variables. We *minimize* a combinational circuit by reducing the number and size of gates that are needed to build it.

minimize

The traditional combinational-circuit-minimization methods that we'll study have as their starting point a truth table or, equivalently, a minterm list or

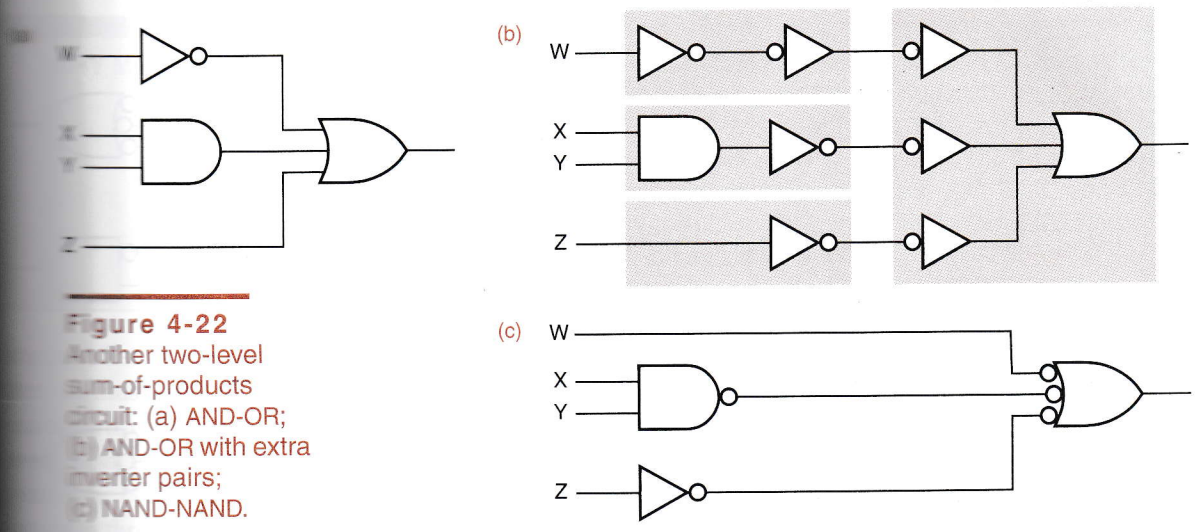


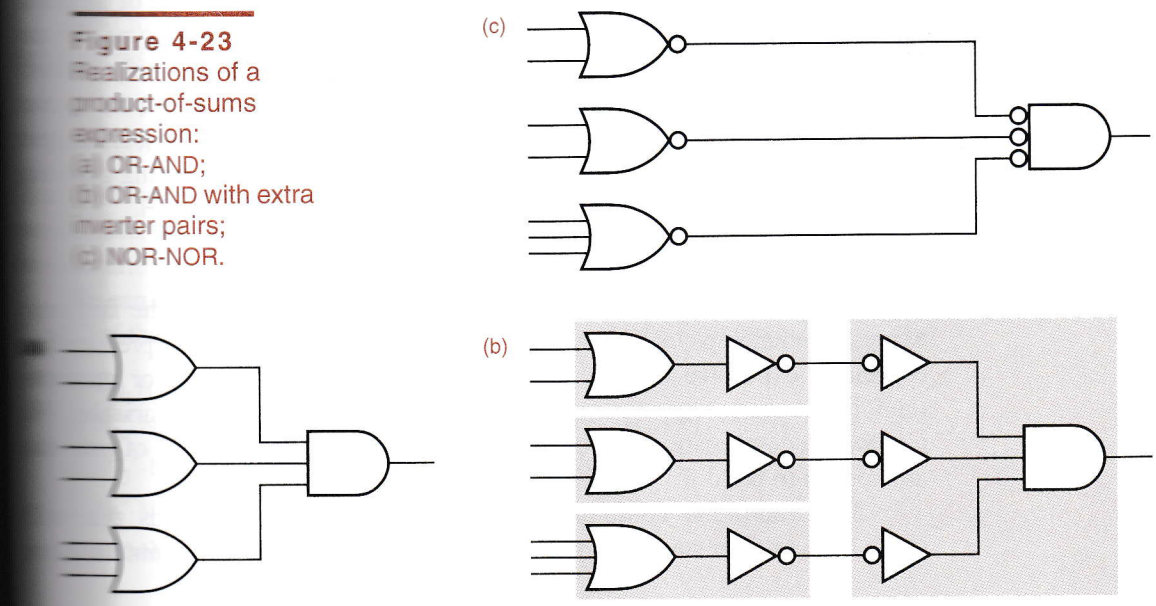
Figure 4-22
Another two-level sum-of-products circuit: (a) AND-OR; (b) AND-OR with extra inverter pairs; (c) NAND-NAND.

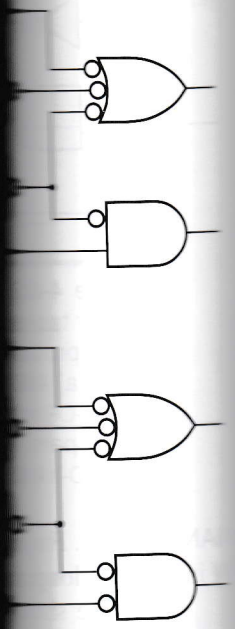
-NAND. For example, Figure 4-22 is an example where an inverter on the is no longer needed, but an inverter must be added to the Z input.

We have shown that any sum-of-products expression can be realized in two ways—as an AND-OR circuit or as a NAND-NAND circuit. The this statement is also true: any product-of-sums expression can be real- as an *OR-AND circuit* or as a *NOR-NOR circuit*. Figure 4-23 shows an ple. Any logic expression can be translated into an equivalent product-of- expression by adding it out, and hence has both OR-AND and NOR-NOR realizations.

OR-AND circuit
NOR-NOR circuit

Figure 4-23
Realizations of a product-of-sums expression: (a) OR-AND; (b) OR-AND with extra inverter pairs; (c) NOR-NOR.





inverter used to
erter placement.

ary logic circuits
and OR gates. After
er, one of the gates
ndard type. We can
uses only standard
to use the inverter
and the bottom gate
inverting gates like
and OR.

from the first logic
duct expressions are
terms or maxterms
er of variables. We
nd size of gates that

methods that we'll
ify, a minterm list or

WHY MINIMIZE? Minimization is an important step in both ASIC design and in PLD-based desig Extra gates and gate inputs require more area in an ASIC chip and thereby increa cost. The number of gates in a PLD is fixed, so you might think that extra gates a free—and they are, until you run out of them and have to upgrade to a bigger, slowe more expensive PLD. Fortunately, most software tools for both ASIC and PL design have a minimization program built in. The purpose of Sections 4.3.3 throug 4.3.7 is to give you a feel for how minimization works.

maxterm list. If we are given a logic function that is not expressed in this form, then we must convert it to an appropriate form before using these methods. For example, if we are given an arbitrary logic expression, then we can evaluate it for every input combination to construct the truth table. The minimization methods reduce the cost of a two-level AND-OR, OR-AND, NAND-NAND, or NOR-NOR circuit in three ways:

1. By minimizing the number of first-level gates.
2. By minimizing the number of inputs on each first-level gate.
3. By minimizing the number of inputs on the second-level gate. This is actually a side effect of the first reduction.

However, the minimization methods do not consider the cost of input inverters; they assume that both true and complemented versions of all input variables are available. While this is not always the case in gate-level or ASIC design, it's very appropriate for PLD-based design; PLDs have both true and complemented versions of all input variables available "for free."

Most minimization methods are based on a generalization of the combining theorems, T10 and T10′:

$$\text{given product term} \cdot Y + \text{given product term} \cdot Y' = \text{given product term}$$

$$(\text{given sum term} + Y) \cdot (\text{given sum term} + Y') = \text{given sum term}$$

That is, if two product or sum terms differ only in the complementing or not of one variable, we can combine them into a single term with one less variable. So we eliminate one gate and the remaining gate has one fewer input.

We can apply this algebraic method repeatedly to combine minterms 1, 3, 5, and 7 of the prime-number detector shown in Figure 4-18 on page 206:

$$\begin{aligned} F &= \Sigma_{N_3,N_2,N_1,N_0}(1, 3, 5, 7, 2, 11, 13) \\ &= N_3' \cdot N_2' \cdot N_1' \cdot N_0 + N_3' \cdot N_2' \cdot N_1 \cdot N_0 + N_3' \cdot N_2 \cdot N_1' \cdot N_0 + N_3' \cdot N_2 \cdot N_1 \cdot N_0 + \ldots \\ &= (N_3' \cdot N_2' \cdot N_1' \cdot N_0 + N_3' \cdot N_2' \cdot N_1 \cdot N_0) + (\cdot N_3' \cdot N_2 \cdot N_1' \cdot N_0 + N_3' \cdot N_2 \cdot N_1 \cdot N_0) + \ldots \\ &= N_3' \cdot N_2' \cdot N_0 + N_3' \cdot N_2 \cdot N_0 + \ldots \\ &= N_3' \cdot N_0 + \ldots \end{aligned}$$

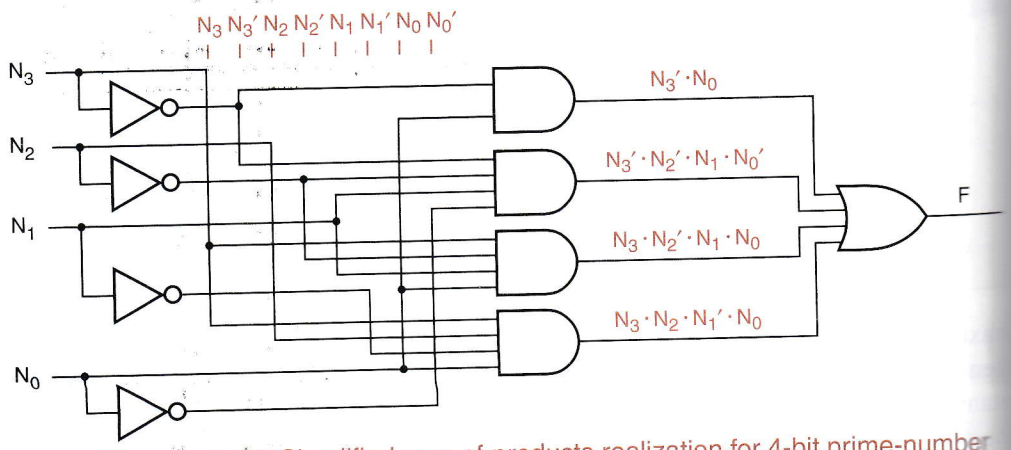


Figure 4-25 Simplified sum-of-products realization for 4-bit prime-number detector.

The resulting circuit is shown in Figure 4-25; it has three fewer gates, and one of the remaining gates has two fewer inputs.

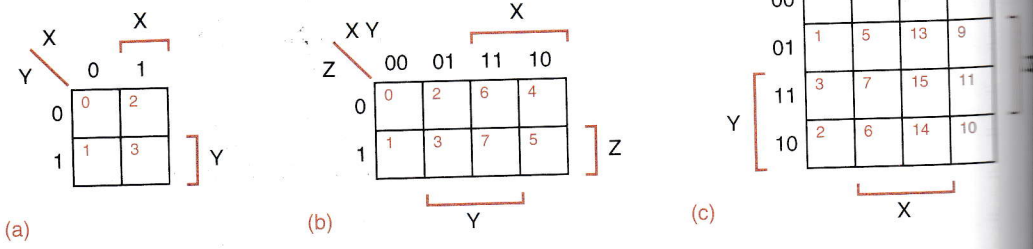
If we had worked a little harder on the preceding expression, we could have saved a couple more first-level gate inputs, though not any more gates. But it's difficult to find terms that can be combined in a jumble of algebraic symbols. In the next subsection we'll begin to explore a minimization method that is more fit for human consumption. Our starting point will be the graphical equivalent of a truth table.

4.3.4 Karnaugh Maps

Karnaugh map

A *Karnaugh map* is a graphical representation of a logic function's truth table. Figure 4-26 shows Karnaugh maps for logic functions of two, three, and four variables. The map for an n-input logic function is an array with 2^n cells, one for each possible input combination or minterm.

Figure 4-26 Karnaugh maps: (a) 2-variable; (b) 3-variable; (c) 4-variable



The rows and columns of a Karnaugh map are labeled so that the input combination for any cell is easily determined from the row and column headings for that cell. The small number inside each cell is the corresponding minterm number in the truth table, assuming that the truth-table inputs are labeled alphabetically from left to right (e.g., X, Y, Z) and the rows are numbered in binary counting order, like all the examples in this text. For example, cell 13 in the 4-variable map corresponds to the truth-table row in which W X Y Z = 1101.

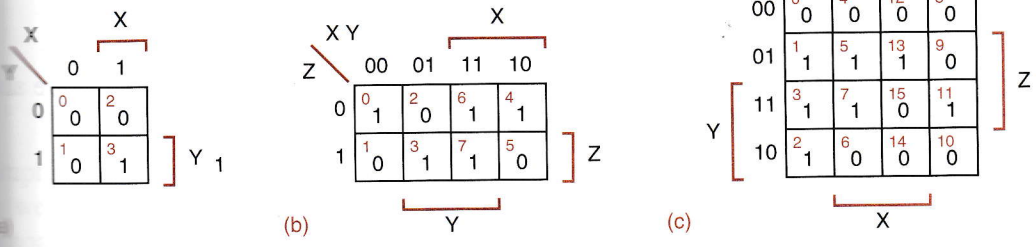
When we draw the Karnaugh map for a given function, each cell of the map contains the information from the like-numbered row of the function's truth table—a 0 if the function is 0 for that input combination, a 1 otherwise.

In this text we use two redundant labelings for map rows and columns. For example, consider the 4-variable map in Figure 4-26(c). The columns are labeled with the four possible combinations of W and X, W X = 00, 01, 11, and 10. Similarly, the rows are labeled with the Y Z combinations. These labels give us all the information we need. However, we also use brackets to associate four regions of the map with the four variables. Each bracketed region is the part of the map in which the indicated variable is 1. Obviously, the brackets convey the same information that is given by the row and column labels.

When we draw a map by hand, it is much easier to draw the brackets than to write out all of the labels. However, we retain the labels in the text's Karnaugh maps as an additional aid to understanding. In any case, you must be sure to label the rows and columns in the proper order to preserve the correspondence between map cells and truth-table row numbers shown in Figure 4-26.

To represent a logic function on a Karnaugh map, we simply copy 1s and 0s from the truth table or equivalent to the corresponding cells of the map. Figure 4-27(a–c) shows Karnaugh maps for $F = \Sigma_{X,Y}(3)$ (a 2-input AND gate), $F = \Sigma_{X,Y,Z}(0, 1, 2, 5, 7)$ (truth table 4-5 on page 197), and $F = \Sigma_{W,X,Y,Z}(1, 2, 3, 5, 7, 11, 15)$ (our 4-bit prime-number detector).

Figure 4-27 Karnaugh map for logic functions: (a) $F = \Sigma_{X,Y}(3)$; (b) $F = \Sigma_{X,Y,Z}(0,3,4,6,7)$; (c) $F = \Sigma_{W,X,Y,Z}(1,2,3,5,7,11,13)$.



In general, we can simplify a logic function by first combining pairs of adjacent 1-cells (minterms) wherever possible and then selecting a set of product terms that covers all of the 1-cells and summing them. Figure 4-28(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 4-29.

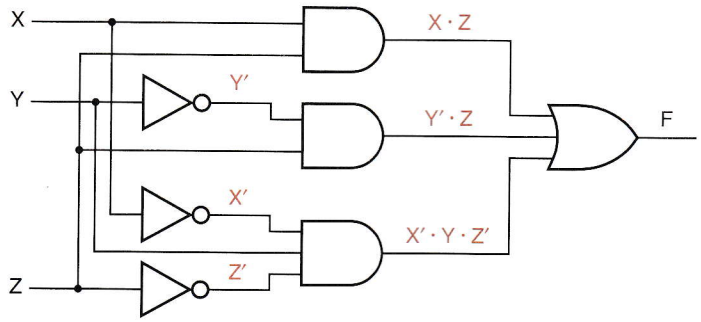


Figure 4-29
Minimized AND-OR circuit.

In many logic functions the cell-combining procedure can be extended to combine more than two 1-cells into a single product term. The number of cells combined will always be a power of 2. For example, consider 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 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

(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

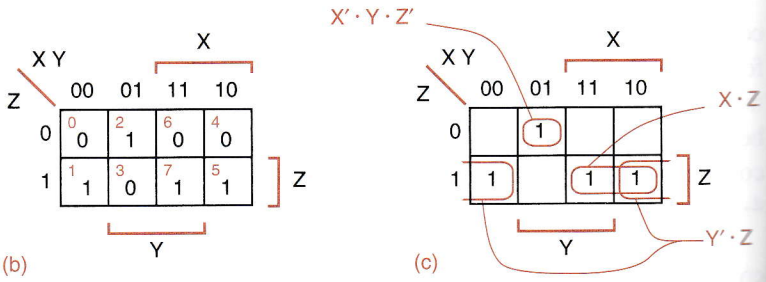


Figure 4-28 $F = \Sigma_{X,Y,Z}(1,2,5,7)$: (a) truth table; (b) Karnaugh map; (c) combining adjacent 1-cells.

Figures 4-28(a) and (b) show the truth table and Karnaugh map for a logic function that we analyzed (beat to death?) in Section 4.2. From now on, we'll reduce the clutter in maps by copying only the 1s or the 0s, not both, as in (c).

4.3.5 Minimizing Sums of Products

By now you must be wondering about the "strange" ordering of the row and column numbers in a Karnaugh map. There is a very important reason for this ordering—each cell corresponds to an input combination that differs from each of its immediately adjacent neighbors in only one variable. For example, cells 5 and 13 in the 4-variable map differ only in the value of W. In the 3- and 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 4-28(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 4-28(b) are also adjacent and can be combined:

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

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 areas of the map where the variable is 0 as well as areas where it is 1, then the variable does not appear in the product term.

Finally, 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 4-30(a) and (b). Here 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 206 can be minimized as shown in Figure 4-31.

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.

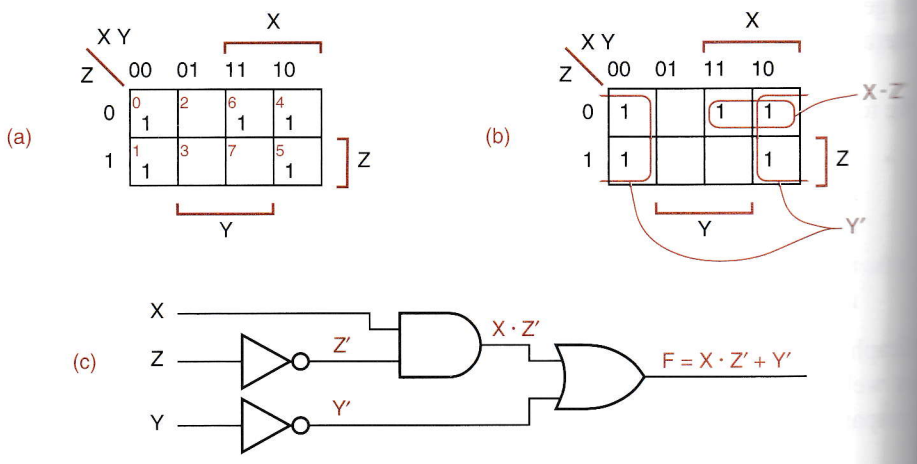


Figure 4-30
$F = \Sigma_{X,Y,Z}(0,1,4,5,6)$: (a) initial Karnaugh map; (b) Karnaugh map with circled product terms; (c) AND/OR circuit.

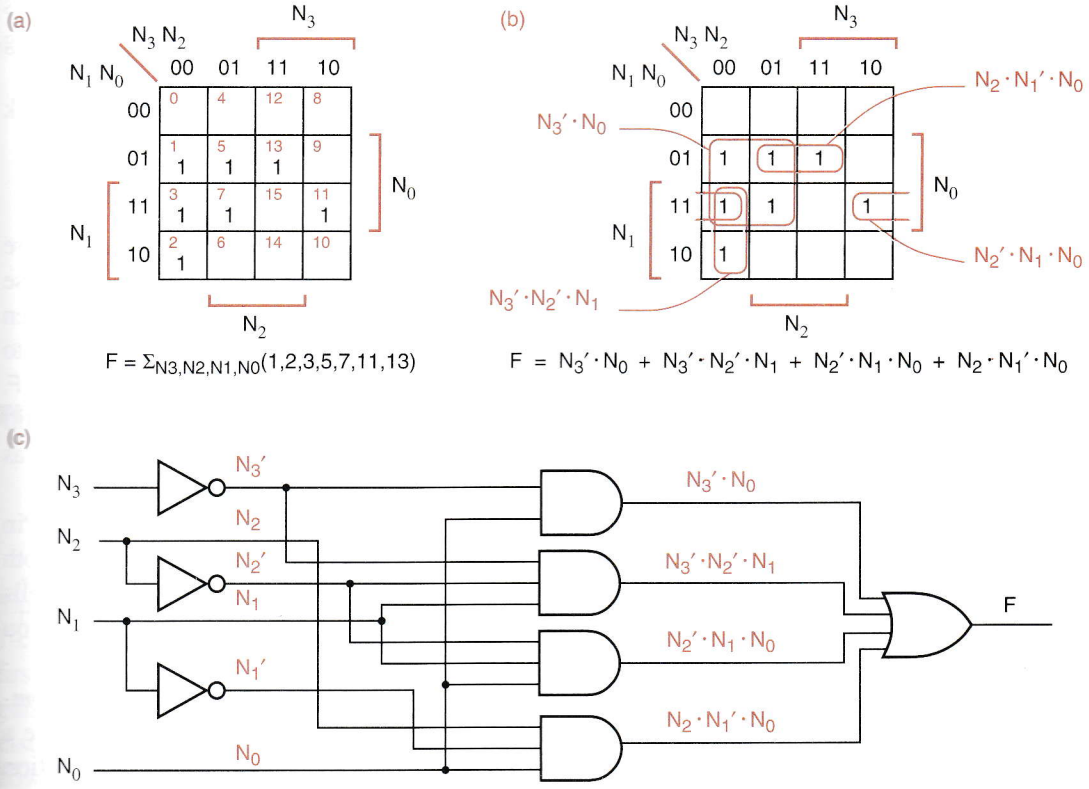


Figure 4-31 Prime-number detector: (a) initial Karnaugh map; (b) circled product terms; (c) minimized circuit.

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 4-31 realizes a minimal sum.

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

- 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. *imply*

That is, if P implies F, then F is 1 for every input combination that P is 1, and maybe some more. We may write the shorthand $P \Rightarrow F$. We may also say that "F *includes* P," or that "F *covers* P." *includes* *covers*

- A *prime implicant* of a logic function $F(X_1,\ldots,X_n)$ is a normal product term $P(X_1,\ldots,X_n)$ that implies F, such that if any variable is removed from P, then the resulting product term does not imply F. *prime implicant*