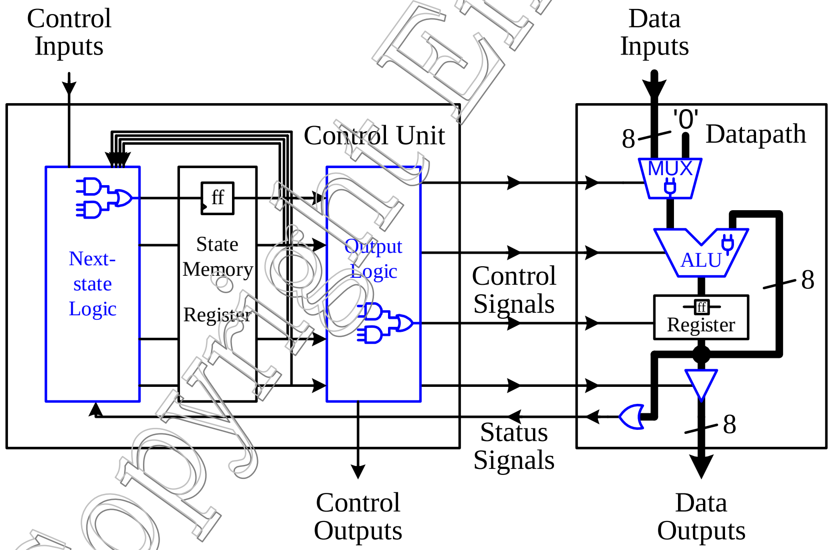
Contents

Chapter 3

Combinational Circuits



Digital circuits, regardless of whether they are part of the control unit or the datapath, are classified as either one of two types: combinational or sequential. **Combinational circuits** are the class of digital circuits where the outputs of the circuit are dependent only on the current inputs. In other words, a combinational circuit is able to produce an output simply from knowing what the current input values are. **Sequential circuits**, on the other hand, are circuits whose outputs are dependent on not only the current inputs, but also on all of the past inputs. Therefore, in order for a sequential circuit to produce an output, it must know the current input and all past inputs. Because of their dependency on past inputs, sequential circuits must contain memory elements in order to remember the history of past input values. Combinational circuits do not need to know the history of past inputs, and therefore, do not require any memory elements. A "large" digital circuit may contain both combinational circuits and sequential circuits. However, regardless of whether it is a combinational circuit or a sequential circuit, it is nevertheless a digital circuit, and so they use the same basic building blocks—the AND, OR, and NOT gates. What makes them different is in the way the gates are connected.

The car security system from Section 2.9 is an example of a combinational circuit. In the example, the siren is turned on when the master switch is on and someone opens the door. If you close the door then the siren will turn off immediately. With this setup, the output, which is the siren, is dependent only on the inputs, which are the master and door switches. For the security system to be more useful, the siren should remain on even after closing the door after it is first triggered. In order to add this new feature to the security system, we need to modify it so that the output is not only dependent on the master and door switches, but also dependent on whether the door has been opened previously or not. A memory element is needed in order to remember whether the door previously was opened or not, and this results in a sequential circuit.

In this and the next chapter, we will look at the design of combinational circuits. In this chapter, we will look at the analysis and design of general combinational circuits. Chapter 4 will look at the design of specific combinational components. Some sample combinational circuits in our microprocessor road map include the next-state logic and output logic in the control unit, and the multiplexer, ALU, comparator, and tri-state buffer in the datapath. We will leave the design of sequential circuits for a later chapter.

In addition to being able to design a functionally correct circuit, we would also like to be able to optimize the circuit in terms of size, speed, and power consumption. Usually, reducing the circuit size will also increase the speed and reduce the power usage. In this chapter, we will look only at reducing the circuit size. Optimizing the circuit for speed and power usage is beyond the scope of this book.

3.1 Analysis of Combinational Circuits

Very often, we are given a digital logic circuit, and we would like to know the operation of the circuit. The analysis of combinational circuits is the process in which we are given a combinational circuit, and we want to derive a precise description of the operation of the circuit. In general, a combinational circuit can be described precisely either with a truth table or with a Boolean function.

3.1.1 Using a Truth Table

For example, given the combinational circuit of Figure 3.1, we want to derive the truth table that describes the circuit. We create the truth table by first listing all of the inputs found in the circuit, one input per column, followed by all of the outputs found in the circuit. Hence, we start with a table with four columns: three columns (*x*, *y*, *z*) for the inputs, and one column (*f*) for the output, as shown in Figure 3.2(a).

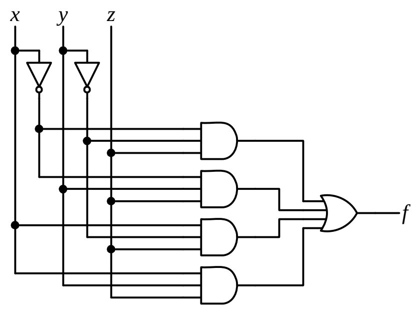


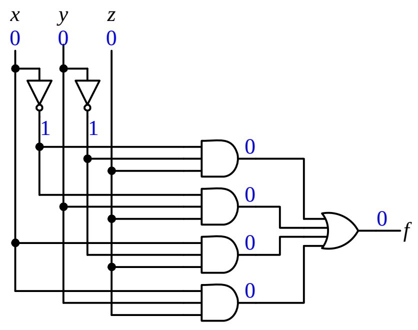
Figure 3.1 Sample combinational circuit.

x	*y*	*z*	*f*

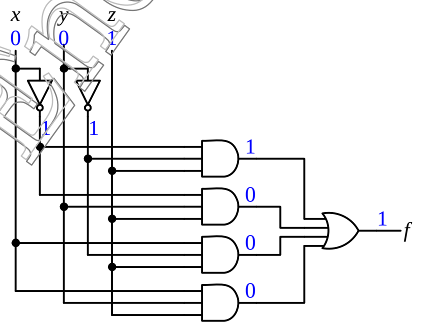
(a)

x	*y*	*z*	*f*
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

(b)



(c)



(d)

x	*y*	*z*	*f*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(e)

Figure 3.2 Deriving the truth table for the sample circuit in Figure 3.1: (a) listing the input and output columns; (b) enumerating all possible combinations of the three input values; (c) circuit annotated with the input values *xyz* = 000; (d) circuit annotated with the input values *xyz* = 001; (e) complete truth table for the circuit.

The next step is to enumerate all possible combinations of 0's and 1's for all of the input variables. In general, for a circuit with *n* inputs, there are 2^n combinations, from 0 to $2^n - 1$. Continuing on with the example, the table in Figure 3.2(b) lists the eight combinations for the three variables in order.

Now, for each row in the table (that is, for each combination of input values), we need to determine what the output value is. This is done by substituting the values for the input variables and tracing through the circuit to the output. For example, using $xyz = 000$, the outputs for all of the AND gates are 0, and ORing all the zeros gives a zero. Therefore, $f = 0$ for this set of values for *x*, *y*, and *z*. This is shown in the annotated circuit in Figure 3.2(c).

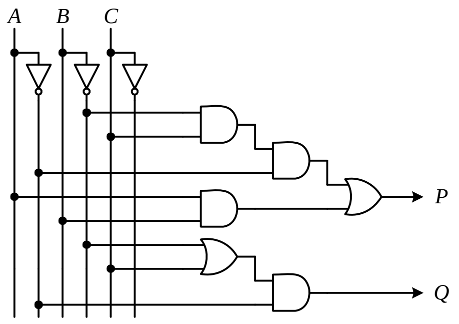
For $xyz = 001$, the output of the top AND gate gives a 1, and 1 ORed with anything gives a 1; therefore, $f = 1$, as shown in the annotated circuit in Figure 3.2(d).

Continuing in this fashion for all of the input combinations, we can complete the final truth table for the circuit, as shown in Figure 3.2(e).

A faster method for evaluating the values for the output signals is to work backwards, that is, to trace the circuit from the output back to the inputs. You want to ask the question: When is the output a 1 (or a 0)? Then trace back to the inputs to see what the input values ought to be in order to get the 1 output. For example, using the circuit in Figure 3.1, *f* is a 1 when any one of the four OR-gate inputs is a 1. For the first input of the OR gate to be a 1, the inputs to the top AND gate must be all 1's. This means that the values for *x*, *y*, and *z* must be 0, 0, and 1, respectively. Repeat this analysis with the remaining three inputs to the OR gate. What you will end up with are the four input combinations for which *f* is a 1. The remaining input combinations, of course, will produce a 0 for *f*.

Example 3.1: Deriving a truth table from a circuit diagram

Derive the truth table for the following circuit with three inputs, *A*, *B* and *C*, and two outputs, *P* and *Q*:



The truth table will have three columns for the three inputs and two columns for the two outputs. Enumerating all possible combinations of the three input values gives eight rows in the table. For each combination of input values, we need to evaluate the output values for both *P* and *Q*. For *P* to be a 1, either of the OR-gate inputs must be a 1. The first input to this OR gate is a 1 if $ABC = 001$. The second input to this OR gate is a 1 if $AB = 11$. Since *C* is not specified in this case, it means that *C* can be either a 0 or a 1. Hence, we get the three input combinations for which *P* is a 1, as shown in the following truth table under the *P* column. The rest of the input combinations will produce a 0 for *P*. For *Q* to be a 1, both inputs of the AND gate must be a 1. Hence, *A* must be a 0, and either *B* is a 0 or *C* is a 1. This gives three input combinations for which *Q* is a 1, as shown in the truth table under the *Q* column.

A	*B*	*C*	*P*	*Q*
0	0	0	0	1
0	0	1	1	1
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	0
1	1	1	1	0

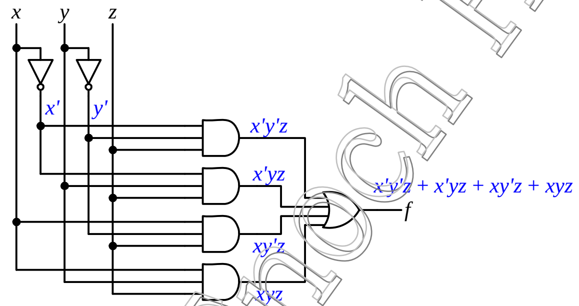
♦

3.1.2 Using a Boolean Function

To derive a Boolean function that describes a combinational circuit, we simply write down the Boolean logical expression at the output of each gate (instead of substituting actual values of 0's and 1's for the inputs) as we trace through the circuit from the primary input to the primary output. Using the sample combinational circuit of Figure 3.1, we note that the logical expression for the output of the top AND gate is *x'y'z*. The logical expressions for the remaining AND gates are, respectively *x'yz*, *xy'z*, and *xyz*. Finally, the outputs from these AND gates are all ORed together. Hence, we get the final expression

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

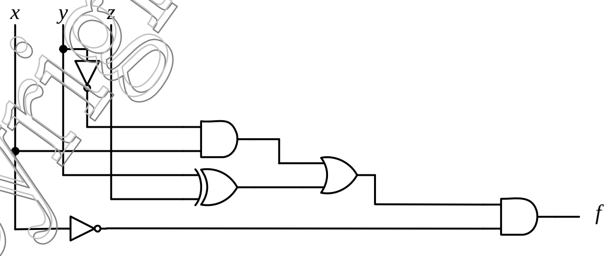
To help keep track of the expressions at the output of each logic gate, we can annotate the outputs of each logic gate with the resulting logical expression as shown here.



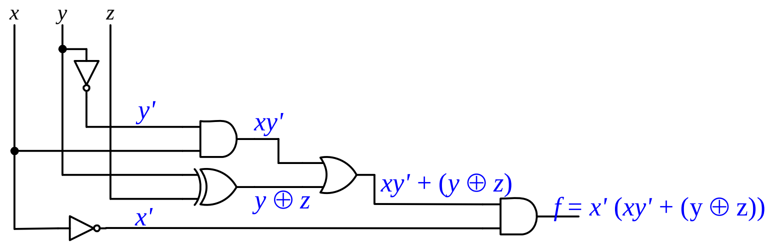
If we substitute all possible combinations of values for all of the variables in the final equation, we should obtain the same truth table as before.

Example 3.2: Deriving a Boolean function from a circuit diagram

Derive the Boolean function for the following circuit with three inputs, *x*, *y*, and *z*, and one output, *f*.



Starting from the primary inputs *x*, *y*, and *z*, we annotate the outputs of each logic gate with the resulting logical expression. Hence, we obtain the annotated circuit:



The Boolean function for the circuit is the final equation $f = x'(xy' + (y \oplus z))$ at the output of the circuit. ♦

If a circuit has two or more outputs, then there must be one equation for each of the outputs. The equations are derived totally independent of each other.

3.2 Synthesis of Combinational Circuits

Synthesis of combinational circuits is just the reverse procedure of the analysis of combinational circuits. In synthesis, we start with a description of the operation of the circuit. From this description, we derive either the truth table or the Boolean logical function that precisely describes the operation of the circuit. Once we have either the truth table or the logical function, we can easily translate that into a circuit diagram.

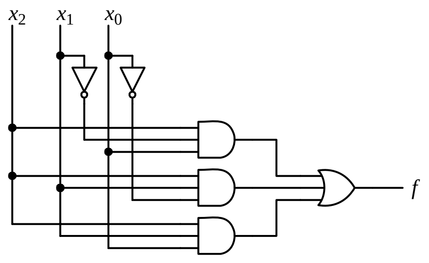
For example, let us construct a 3-bit comparator circuit that outputs a 1 if the number is greater than or equal to 5 and outputs a 0 otherwise. In other words, construct a circuit that outputs a 0 if the input is a number between 0 and 4 (inclusive) and outputs a 1 if the input is a number between 5 and 7 (inclusive). The reason why the maximum number is 7 is because the range for an unsigned 3-bit binary number is from 0 to 7. Hence, we can use the three bits, x_2, x_1, and x_0, to represent the 3-bit input value to the comparator. From the description, we obtain the following truth table:

Decimal	Binary number			Output
number	x_2	x_1	x_0	f
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

In constructing the circuit, we are interested only in when the output is a 1 (i.e., when the function f is a 1). Thus, we only need to consider the rows where the output function $f = 1$. From the previous truth table, we see that there are three rows where $f = 1$, which give the three AND terms $x_2x_1'x_0$, $x_2x_1x_0'$, and $x_2x_1x_0$. Notice that the variables in the AND terms are such that it is inverted if its value is a 0, and not inverted if its value is a 1. In the case of the first AND term, we want $f = 1$ when $x_2 = 1$ and $x_1 = 0$ and $x_0 = 1$; and this is satisfied in the expression $x_2x_1'x_0$. Similarly, the second and third AND terms are satisfied in the expressions $x_2x_1x_0'$ and $x_2x_1x_0$, respectively. Finally, we want $f = 1$ when either one of these three AND terms is equal to 1. So we ORed the three AND terms together, giving us our final expression:

$$f = x_2x_1'x_0 + x_2x_1x_0' + x_2x_1x_0 \qquad (3.1)$$

In drawing the schematic diagram, we simply convert the AND operators to AND gates, OR operators to OR gates, and primes to NOT gates. The equation is in the sum-of-products format, meaning that it is summing (ORing) the product (AND) terms. A sum-of-products equation translates to a two-level circuit with the first level being made up of AND gates and the second level made up of OR gates. Each of the three AND terms contains three variables, so we use a 3-input AND gate for each of the three AND terms. The three AND terms are ORed together, so we use a 3-input OR gate to connect the output of the three AND gates. For each inverted variable, we need an inverter. The schematic diagram derived from Equation 3.1 is shown here.



From this discussion we see that any combinational circuit can be constructed using only AND, OR, and NOT gates from either a truth table or a Boolean equation.

Example 3.3: Synthesizing a combinational circuit from a truth table

Synthesize a combinational circuit from the following truth table. The three variables, *a*, *b*, and *c*, are input signals, and the two variables, *x*, and *y*, are output signals.

a	*b*	*c*	*x*	*y*
0	0	0	1	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	1
1	1	0	1	0
1	1	1	0	0

We either can derive the Boolean equation from the truth table and then derive the circuit from the equation, or we can derive the circuit directly from the truth table. For this example, we first will derive the Boolean equation. Since there are two output signals, there will be two equations; one for each output signal.

From Section 2.6, we saw that a function is formed by summing its 1-minterms. For output *x*, there are five 1-minterms: m_0, m_2, m_3, m_5, and m_6. These five minterms represent the five AND terms, *a'b'c'*, *a'bc'*, *a'bc*, *ab'c*, and *abc'*. Hence, the equation for *x* is

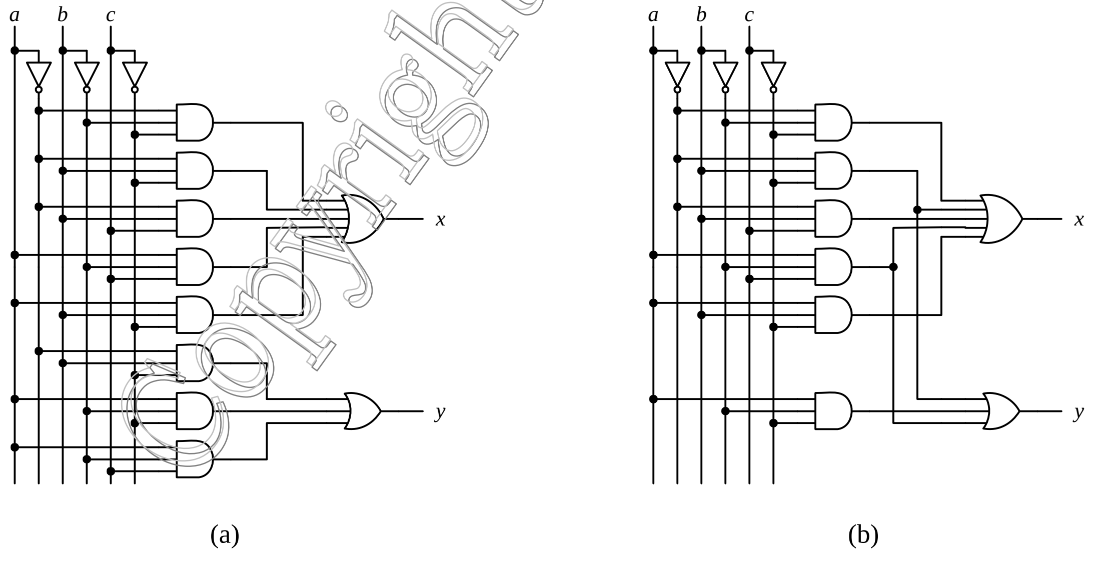
$$x = a'b'c' + a'bc' + a'bc + ab'c + abc'$$

Similarly, the output signal *y* has three 1-minterms, and they are *a'bc'*, *ab'c'*, and *ab'c*. Hence, the equation for *y* is

$$y = a'bc' + ab'c' + ab'c$$

The combinational circuit constructed from these two equations is shown in Figure 3.3(a). Each 3-variable AND term is replaced by a 3-input AND gate. The three inputs to these AND gates are connected to the three input variables *a*, *b*, and *c*, either directly if the variable is not primed or through a NOT gate if the variable is primed. For output *x*, a 5-input OR gate is used to connect the outputs of the five AND gates for the corresponding five AND terms. For output *y*, a 3-input OR gate is used to connect the outputs of the three AND gates.

Notice that the two AND terms, *a'bc'*, and *ab'c*, appear in both the *x* and the *y* equations. As a result, we do not need to generate these two signals twice. Hence, we can reduce the size of the circuit by not duplicating these two AND gates, as shown in Figure 3.3(b). ♦



(a) (b)

Figure 3.3 Combinational circuit for Example 3.3: (a) no reduction; (b) with reduction.

3.3 * Technology Mapping

To reduce implementation cost and turnaround time to produce a digital circuit on an IC, designers often make use of off-the-shelf semi-custom gate arrays. Many gate arrays are ICs that have only NAND gates or NOR gates built in them, but their input and output connections are not yet connected. To use these gate arrays, a designer simply has to specify where to make these connections between the gates. The problem here is that, when we use these gate arrays to implement a circuit, we need to convert all AND, OR, and NOT gates in the circuit to use only NAND or NOR gates, depending on what is available in the gate array. In addition, these NAND and NOR gates usually have the same number of fixed inputs, for example, only three inputs.

In Section 3.2, we saw that any combinational circuit can be constructed with only AND, OR, and NOT gates. It turns out that any combinational circuit can also be constructed with either only NAND gates or only NOR gates. The reason why we want to use only NAND or NOR gates will be made clear when we look at how these gates are built at the transistor level in Chapter 5. We will now look at how a circuit with AND, OR, and NOT gates is converted to one with only NAND or only NOR gates.

The conversion of any given circuit to use only 2-input NAND or 2-input NOR gates is possible by observing the following equalities. These equalities, in fact, are obtained from the Boolean algebra theorems from Chapter 2.

Rule 1:	$x'' = x$	(double NOT)
Rule 2:	$x' = (x \bullet x)' = (x \bullet 1)'$	(NOT to NAND)
Rule 3:	$x' = (x + x)' = (x + 0)'$	(NOT to NOR)
Rule 4:	$xy = ((xy)')'$	(AND to NAND)
Rule 5:	$x + y = ((x + y)')' = (x' y')'$	(OR to NAND)
Rule 6:	$xy = ((xy)')' = (x' + y')'$	(AND to NOR)
Rule 7:	$x + y = ((x + y)')'$	(OR to NOR)

Rule 1 simply says that a double inverter can be eliminated altogether. Rules 2 and 3 convert a NOT gate to a NAND gate or a NOR gate, respectively. For both Rules 2 and 3, there are two ways to convert a NOT gate to either a NAND gate or a NOR gate. For the first method, the two inputs are connected in common. For the second method, one input is connected to the logic 1 for the NAND gate and to 0 for the NOR gate. Rule 4 applies Rule 1 to the AND gate. The resulting expression gives us a NAND gate followed by a NOT gate. We can then use Rule 2 to change the NOT gate to a NAND gate. Rule 5 changes an OR gate to use two NOT gates and a NAND gate by first applying Rule 1 and then De Morgan's theorem. Again, the two NOT gates can be changed to two NAND gates using Rule 2. Similarly, Rule 6 converts an AND gate to use two NOT gates and a NOR gate, and Rule 7 converts an OR gate to a NOR gate followed by a NOT gate.

In a circuit diagram, these rules are translated to the equivalent circuits, as shown in Figure 3.4. Rules 2, 4, and 5 are used if we want to convert a circuit to use only 2-input NAND gates; whereas, Rules 3, 6, and 7 are used if we want to use only 2-input NOR gates.

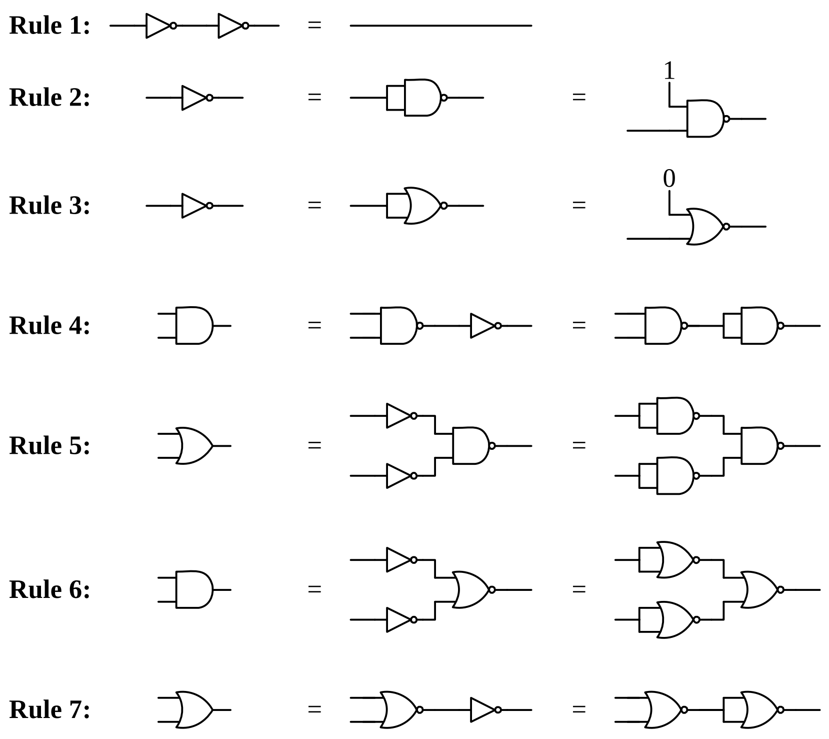


Figure 3.4 Circuits for converting from AND, OR, and NOT gates to NAND or NOR gates.

Another thing that we might want is to get the functionality of a 2-input NAND or 2-input NOR gate from a 3-input NAND or 3-input NOR gate, respectively. In other words, we want to use a 3-input NAND or NOR gate to work like a 2-input NAND or NOR gate, respectively. On the other hand, we might also want to get the reverse of that (that is, to get the functionality of a 3-input NAND or 3-input NOR gate from a 2-input NAND or 2-input NOR gate, respectively). These equalities are shown in the following rules and their corresponding circuits in Figure 3.5.

Rule 8: $(x \bullet y)' = (x \bullet y \bullet y)'$ (2-input to 3-input NAND)

Rule 9: $(x + y)' = (x + y + y)'$ (2-input to 3-input NOR)

Rule 10: $(abc)' = ((ab)\, c)' = ((ab)''\, c)'$ (3-input to 2-input NAND)

Rule 11: $(a+b+c)' = ((a+b) + c)' = ((a+b)'' + c)'$ (3-input to 2-input NOR)

Rule 8 converts from a 2-input NAND gate to a 3-input NAND gate. Rule 9 converts from a 2-input NOR gate to a 3-input NOR gate. Rule 10 converts from a 3-input NAND gate to using only 2-input NAND gates. Rule 11 converts from a 3-input NOR gate to using only 2-input NOR gates. Notice that for Rules 10 and 11, an extra NOT gate is needed in between the two gates.

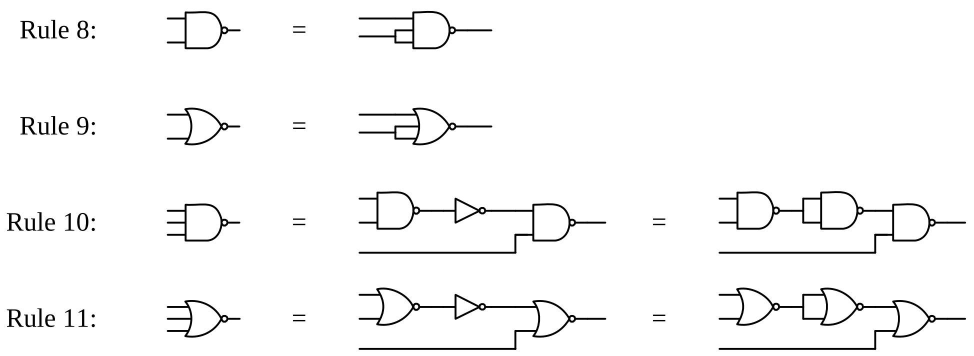
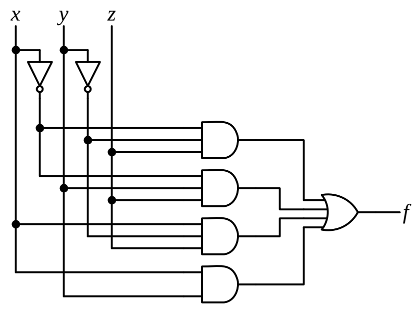


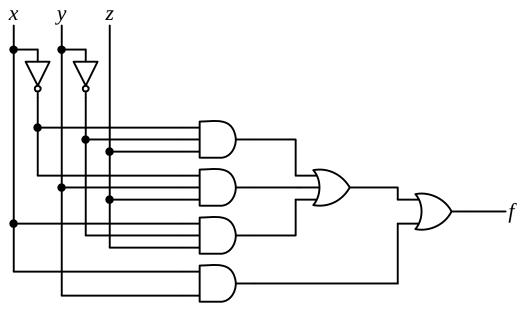
Figure 3.5 Circuits for converting 2-input to 3-input NAND or NOR gate and vice versa.

Example 3.4: Converting a circuit to use only 3-input NAND gates

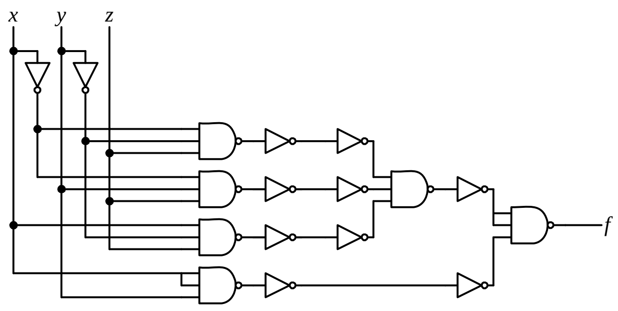
Convert the following circuit to use only 3-input NAND gates.



First, we need to change the 4-input OR gate to a 3- and 2-input OR gates.

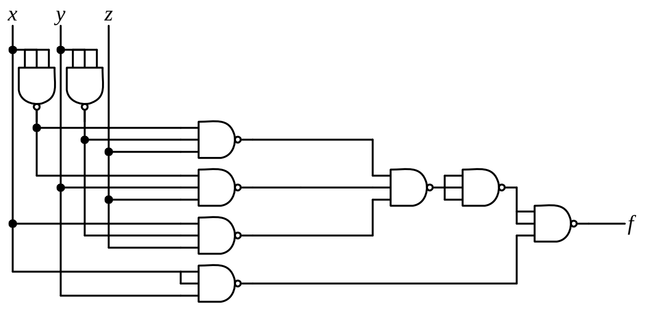


Then we will use Rule 4 to change all of the AND gates to 3-input NAND gates with inverters and Rule 5 to change all of the OR gates to 3-input NAND gates with inverters. The 2-input NAND gates are replaced with 3-input NAND gates with two of its inputs connected together.



add dots

Finally, we eliminate all the double inverters and replace the remaining inverters with NAND gates with their inputs connected together.



♦

3.4 Minimization of Combinational Circuits

When constructing digital circuits, in addition to obtaining a functionally correct circuit, we like to optimize them in terms of circuit size, speed, and power consumption. In this section, we will focus on the reduction of circuit size. Usually, by reducing the circuit size, we will also improve on speed and power consumption. We have seen in the previous sections that any combinational circuit can be represented using a Boolean function. The size of the circuit is proportional directly to the size or complexity of the functional expression. In fact, it is a one-to-one correspondence between the functional expression and the circuit size. In Section 2.5.1, we saw how we can transform a Boolean function to another equivalent function by using the Boolean algebra theorems. If the resulting function is simpler than the original, then we want to implement the circuit based on the simpler function, since that will give us a smaller circuit size.

Using Boolean algebra to transform a function to one that is simpler is not an easy task, especially for the computer. There is no formula that says which is the next theorem to use. Luckily, there are easier methods for reducing Boolean functions. The **Karnaugh map** method is an easy way for reducing an equation manually and is discussed in Section 3.4.1. The **Quine-McCluskey** or **tabulation** method for reducing an equation is ideal for programming the computer and is discussed in Section 3.4.3.

3.4.1 Karnaugh Maps

To minimize a Boolean equation in the sum-of-products form, we need to reduce the number of product terms by applying the Combining Boolean theorem (Theorem 14) from Section 2.5.1. In so doing, we also will have reduced the number of variables used in the product terms. For example, given the following 3-variable function:

$$F = xy'z' + xyz'$$

we can factor out the two common variables xz' and reduce it to

$$\begin{aligned} F &= xz'\,(y' + y) \\ &= xz'\,1 \\ &= xz' \end{aligned}$$

In other words, two product terms that differ by only one variable, whose value is a 0 (primed) in one term and a 1 (unprimed) in the other term, can be combined together to form just one term with that variable omitted, as shown in the previous equations. Thus, we have reduced the number of product terms, and the resulting product term has one less variable. By reducing the number of product terms, we reduce the number of OR operators required, and by reducing the number of variables in a product term, we reduce the number of AND operators required.

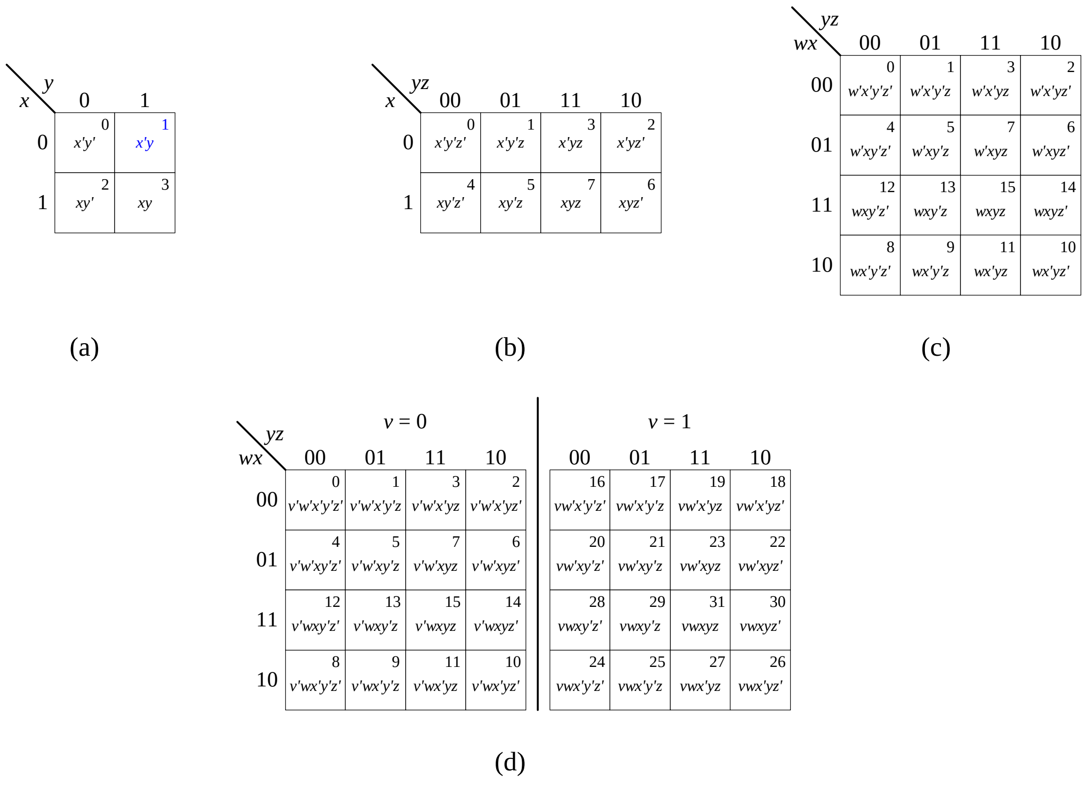
Looking at a logic function's truth table, sometimes it is difficult to see how the product terms can be combined and minimized. A **Karnaugh map** (**K-map** for short) provides a simple and straightforward procedure for combining these product terms. A K-map is just a graphical representation of a logic function's truth table, where the minterms are grouped in such a way that it allows one to easily see which of the minterms can be combined. The

K-map is a two-dimensional array of squares, each of which represents one minterm in the Boolean function. Thus, the map for an *n*-variable function is an array with 2^n squares.

Figure 3.6 shows the K-maps for functions with 2, 3, 4, and 5 variables. Notice the labeling of the columns and rows are such that any two adjacent columns or rows differ in only one bit change. This condition is required because we want minterms in adjacent squares to differ in the value of only one variable or one bit, and so these minterms can be combined together. This is why the labeling for the third and fourth columns and for the third and fourth rows are always interchanged. When we read K-maps, we need to visualize them as such that the two end columns or rows wrap around, so that the first and last columns and the first and last rows are really adjacent to each other, because they also differ in only one bit.

In Figure 3.6, the K-map squares are annotated with their minterms and minterm numbers for easy reference only. For example, in Figure 3.6(a) for a 2-variable K-map, the entry in the first row and second column is labeled *x'y* and annotated with the number 1. This is because the first row is when the variable *x* is a 0, and the second column is when the variable *y* is a 1. Since, for minterms, we need to prime a variable whose value is a 0 and not prime it if its value is a 1, this entry represents the minterm *x'y*, which is minterm number 1. Be careful that, if we label the rows and columns differently, the minterms and the minterm numbers will be in different locations. When we use K-maps to minimize an equation, we will not write these in the squares. Instead, we will be putting 0's and 1's in the squares.

For a 5-variable K-map, as shown in Figure 3.6(d), we need to visualize the right half of the array (where *v* = 1) to be on top of the left half (where *v* = 0). In other words, we need to view the map as three-dimensional. Hence, although the squares for minterms 2 and 16 are located next to each other, they are not considered to be adjacent to each other. On the other hand, minterms 0 and 16 are adjacent to each other, because one is on top of the other.



(a)

x \ y	0	1
0	0 x'y'	1 x'y
1	2 xy'	3 xy

(b)

x \ yz	00	01	11	10
0	0 x'y'z'	1 x'y'z	3 x'yz	2 x'yz'
1	4 xy'z'	5 xy'z	7 xyz	6 xyz'

(c)

wx \ yz	00	01	11	10
00	0 w'x'y'z'	1 w'x'y'z	3 w'x'yz	2 w'x'yz'
01	4 w'xy'z'	5 w'xy'z	7 w'xyz	6 w'xyz'
11	12 wxy'z'	13 wxy'z	15 wxyz	14 wxyz'
10	8 wx'y'z'	9 wx'y'z	11 wx'yz	10 wx'yz'

(d)

	v = 0				v = 1			
wx \ yz	00	01	11	10	00	01	11	10
00	0 v'w'x'y'z'	1 v'w'x'y'z	3 v'w'x'yz	2 v'w'x'yz'	16 vw'x'y'z'	17 vw'x'y'z	19 vw'x'yz	18 vw'x'yz'
01	4 v'w'xy'z'	5 v'w'xy'z	7 v'w'xyz	6 v'w'xyz'	20 vw'xy'z'	21 vw'xy'z	23 vw'xyz	22 vw'xyz'
11	12 v'wxy'z'	13 v'wxy'z	15 v'wxyz	14 v'wxyz'	28 vwxy'z'	29 vwxy'z	31 vwxyz	30 vwxyz'
10	8 v'wx'y'z'	9 v'wx'y'z	11 v'wx'yz	10 v'wx'yz'	24 vwx'y'z'	25 vwx'y'z	27 vwx'yz	26 vwx'yz'

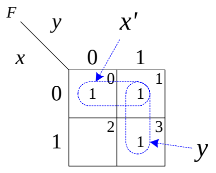
Figure 3.6 Karnaugh maps for: (a) 2 variables; (b) 3 variables; (c) 4 variables; (d) 5 variables.

Given a Boolean function, we set the value for each K-map square to either a 0 or a 1, depending on whether that minterm for the function is a 0-minterm or a 1-minterm, respectively. However, since we are only interested in using the 1-minterms for a function, the 0's are sometimes not written in the 0-minterm squares.

For example, the K-map for the 2-variable function:

$$F = x'y' + x'y + xy$$

is



The 1-minterms, m_0 ($x'y'$) and m_1 ($x'y$), are adjacent to each other, which means that they differ in the value of only one variable. In this case, x is 0 for both minterms, but for y, it is a 0 for one minterm and a 1 for the other minterm. Thus, variable y can be dropped, and the two terms are combined together giving just x'. The prime in x' is because x is 0 for both minterms. This reasoning corresponds with the expression:

$$x'y' + x'y = x' (y' + y) = x' (1) = x'$$

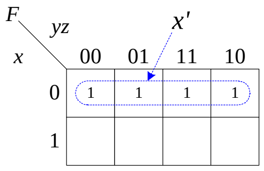
Similarly, the 1-minterms m_1 ($x'y$) and m_3 (xy) are also adjacent, and y is the variable having the same value for both minterms, and so they can be combined to give

$$x'y + xy = (x' + x)\, y = (1)\, y = y$$

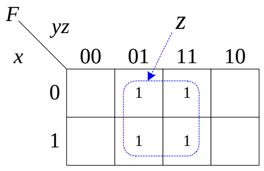
We use the term **subcube** to refer to a rectangle of adjacent 1-minterms. These subcubes must be rectangular in shape and can only have sizes that are powers of two. Formally, for an n-variable K-map, an *m-subcube* is defined as that set of 2^m minterms in which $n - m$ of the variables will have the same value in every minterm, while the remaining variables will take on the 2^m possible combinations of 0's and 1's. Thus, a 1-minterm all by itself is called a 0-subcube, two adjacent 1-minterms is called a 1-subcube, and so on. In the previous 2-variable K-map, there are two 1-subcubes: one labeled with x' and one labeled with y.

A 2-subcube will have four adjacent 1-minterms and can be in the shape of any one of those shown in Figure 3.7(a) through (e). Notice that Figure 3.7(d) and (e) also form 2-subcubes, even though the four 1-minterms are not physically adjacent to each other. They are considered to be adjacent because the first and last rows and the first and last columns wraparound in a K-map. In Figure 3.7(f), the four 1-minterms cannot form a 2-subcube, because even though they are physically adjacent to each other, they do not form a rectangle. However, they can form three 1-subcubes—$y'z$, $x'y'$ and $x'z$.

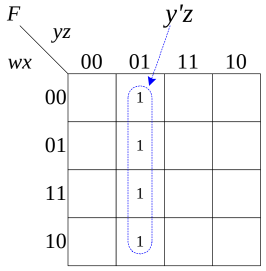
We say that a subcube is *characterized* by the variables having the same values for all of the minterms in that subcube. In general, an m-subcube for an n-variable K-map will be characterized by $n - m$ variables. If the value that is similar for all of the variables is a 1, that variable is unprimed; whereas, if the value that is similar for all of the variables is a 0, that variable is primed. In an expression, this is equivalent to the resulting smaller product term when the minterms are combined together. For example, the 2-subcube in Figure 3.7(d) is characterized by z', since the value of z is 0 for all of the minterms, whereas the values for x and y are not all the same for all of the minterms. Similarly, the 2-subcube in Figure 3.7(e) is characterized by $x'z'$.



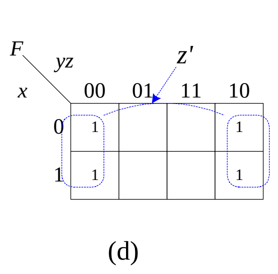
(a)

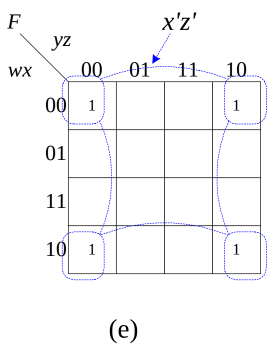


(b)



(c)





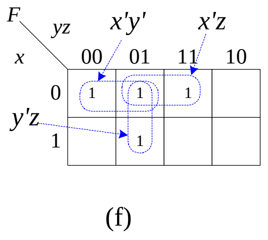


Figure 3.7 Examples of K-maps with 2-subcubes: (a) and (b) 3-variable; (c) 4-variable; (d) 3-variable with wraparound subcube; (e) 4-variable with wraparound subcube; (f) four adjacent minterms that cannot form a 2-subcube.

For a 5-variable K-map, as shown in Figure 3.8, we need to visualize the right half of the array (where $v = 1$) to be on top of the left half (where $v = 0$). Thus, for example, minterm 20 is adjacent to minterm 4 since one is on top of the other, and they form the 1-subcube $w'xy'z'$. Even though minterm 6 is physically adjacent to minterm 20 on the map, they cannot be combined together, because when you visualize the right half as being on top of the left half, then they really are not on top of each other. Instead, minterm 6 is adjacent to minterm 4 because the columns wrap around, and they form the subcube $v'w'xz'$. Minterms 9, 11, 13, 15, 25, 27, 29, and 31 are all adjacent, and together they form the subcube wz. Now that we are viewing this 5-variable K-map in three dimensions, we also need to change the condition of the subcube shape to be a three-dimensional rectangle.

You can see that this visualization becomes almost impossible to work with very quickly as we increase the number of variables. In more realistic designs with many more variables, tabular methods instead of K-maps are used for reducing the size of equations.

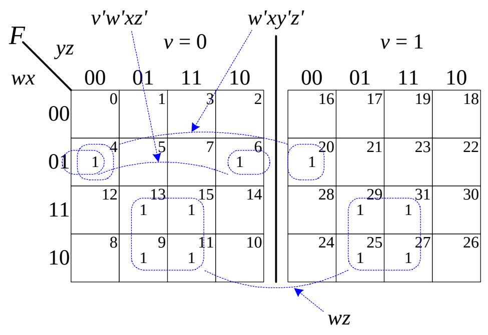


Figure 3.8 A 5-variable K-map with three wraparound subcubes.

The K-map method reduces a Boolean function from its canonical form to its standard form. The goal for the K-map method is to find as few subcubes as possible to cover all of the 1-minterms in the given function. This naturally implies that the size of the subcube should be as big as possible. The reasoning for this is that each subcube corresponds to a product term, and all of the subcubes (or product terms) must be ORed together to get the function. Larger subcubes require fewer AND gates because of fewer variables in the product term, and fewer subcubes will require fewer inputs to the OR gate.

The procedure for using the K-map method is as follows:

1. Draw the appropriate K-map for the given function and place a 1 in the squares that correspond to the function's 1-minterms.

2. For each 1-minterm, find the largest subcube that covers this 1-minterm. This largest subcube is known as a

prime implicant (PI). By definition, a **prime implicant** is a subcube that is not contained within any other subcube. If there is more than one subcube that is of the same size as the largest subcube, then they are all prime implicants.

3. Look for 1-minterms that are covered by only one prime implicant. Since this prime implicant is the only subcube that covers this particular 1-minterm, this prime implicant must be in the final solution. This prime implicant is referred to as an *essential* prime implicant (EPI). By definition, an **essential prime implicant** is a prime implicant that includes a 1-minterm that is not included in any other prime implicant.

4. Create a minimal cover list by selecting the smallest possible number of prime implicants such that every 1-minterm is contained in at least one prime implicant. This cover list must include all of the essential prime implicants plus zero or more of the remaining prime implicants. It is acceptable that a particular 1-minterm is covered in more than one prime implicant, but all 1-minterms must be covered.

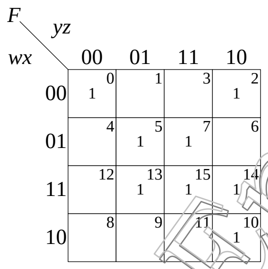
5. The final minimized function is obtained by ORing all of the prime implicants from the minimal cover list.

Note that the final minimized function obtained by the K-map method may not be in its most reduced form. It is only in its most reduced *standard* form. Sometimes, it is possible to reduce the standard form further into a non-standard form.

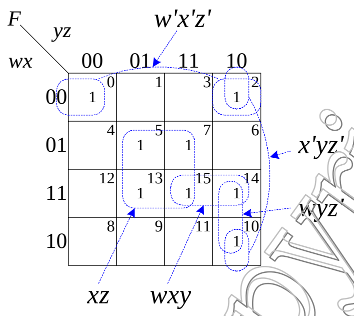
Example 3.5: Using K-map to minimize a 4-variable function

Use the K-map method to minimize a 4-variable (w, x, y, and z) function F with the 1-minterms: m_0, m_2, m_5, m_7, m_{10}, m_{13}, m_{14}, and m_{15}.

We start with the following 4-variable K-map with a 1 placed in each of the eight minterm squares:



The prime implicants for each of the 1-minterms are shown in the following K-map and table:



1-minterm	Prime Implicant
m_0	$w'x'z'$
m_2	$w'x'z'$, $x'yz'$
m_5	xz
m_7	xz
m_{10}	$x'yz'$, wyz'
m_{13}	xz
m_{14}	wyz', wxy
m_{15}	xz

For minterm m_0, there is only one prime implicant $w'x'z'$. For minterm m_2, there are two 1-subcubes that cover it, and they are the largest. Therefore, m_2 has two prime implicants, $w'x'z'$ and $x'yz'$. When we consider m_{14}, again there are two 1-subcubes that cover it, and they are the largest. So m_{14} also has two prime implicants. Minterm m_{15}, however, has only one prime implicant, xz. Although the 1-subcube wxy also covers m_{15}, it is not a prime implicant for m_{15} because it is smaller than the 2-subcube xz.

From the K-map, we see that there are five prime implicants: *w'x'z'*, *x'yz'*, *xz*, *wyz'*, and *wxy*. Of these five prime implicants, *w'x'z'* and *xz* are essential prime implicants, since m_0 is covered only by *w'x'z'*, and m_5, m_7, and m_{13} are covered only by *xz*.

We start the cover list by including the two essential prime implicants *w'x'z'* and *xz*. These two subcubes will have covered the minterms m_0, m_2, m_5, m_7, m_{13}, and m_{15}. To cover the remaining two uncovered minterms, m_{10} and m_{14}, we want to use as few prime implicants as possible. Hence, we select the prime implicant *wyz'*, which covers both of them.

Finally, our reduced standard-form equation is obtained by ORing the two essential prime implicants and one prime implicant in the cover list:

$$F = w'x'z' + xz + wyz'$$

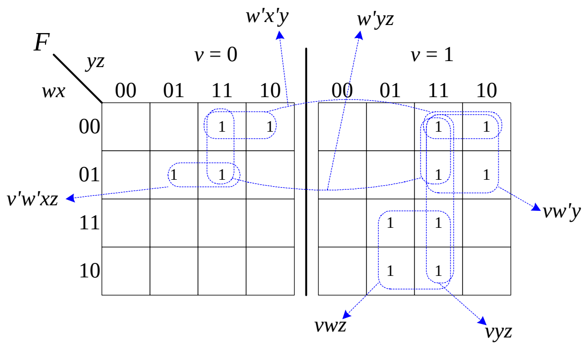
Notice that we can reduce this standard form equation even further by factoring out the *z'* from the first and last term to get the nonstandard-form equation

$$F = z' (w'x' + wy) + xz$$ ♦

Example 3.6: Using K-map to minimize a 5-variable function

Use the K-map method to minimize a 5-variable function *F* (*v*, *w*, *x*, *y* and *z*) with the 1-minterms: *v'w'x'yz'*, *v'w'x'yz*, *v'w'xy'z*, *v'w'xyz*, *vw'x'yz'*, *vw'x'yz*, *vw'xyz'*, *vw'xyz*, *vwx'y'z*, *vwx'yz*, *vwxy'z*, and *vwxyz*.

First, we obtain the following K-map.



The list of prime implicants is: *v'w'xz*, *w'x'y*, *w'yz*, *vw'y*, *vyz*, and *vwz*. From this list of prime implicants, *w'yz* and *vyz* are not essential. The four remaining essential prime implicants are able to cover all of the 1-minterms. Hence, the solution in standard form is

$$F = v'w'xz + w'x'y + vw'y + vwz$$ ♦

3.4.2 Don't-cares

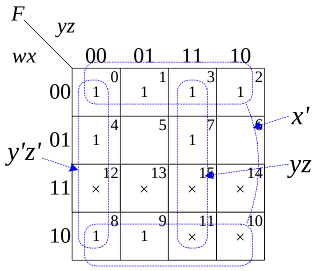
There are times when a function is not specified fully. In other words, there are some minterms for the function where we do not care whether their values are a 0 or a 1. When drawing the K-map for these "**don't-care**" minterms, we assign an "×" in that square instead of a 0 or a 1. Usually, a function can be reduced even further if we remember that these ×'s can be either a 0 or a 1. As you recall when drawing K-maps, enlarging a subcube reduces the number of variables for that term. Thus, in drawing subcubes, some of them may be enlarged if we treat some of these ×'s as 1's. On the other hand, if some of these ×'s will not enlarge a subcube, then we want to treat them as 0's so that we do not need to cover them. It is not necessary to treat all ×'s to be all 1's or all 0's. We can assign some ×'s to be 0's and some to be 1's.

For example, given a function having the following 1-minterms and don't-care minterms:

1-minterms: m_0, m_1, m_2, m_3, m_4, m_7, m_8, and m_9

×-minterms: m_{10}, m_{11}, m_{12}, m_{13}, m_{14}, and m_{15}

we obtain the following K-map with the prime implicants *x'*, *yz*, and *y'z'*.



Notice that, in order to get the 4-subcube characterized by *x'*, the two don't-care minterms, m_{10} and m_{11}, are taken to have the value 1. Similarly, the don't-care minterms, m_{12} and $m_{15,}$ are assigned a 1 for the subcubes *y'z'* and *yz*, respectively. On the other hand, the don't-care minterms, m_{13} and m_{14}, are taken to have the value 0, so that they do not need to be covered in the solution. The reduced standard form function as obtained from the K-map is, therefore,

$$F = x' + yz + y'z'$$

Again, this equation can be reduced further by recognizing that $yz + y'z' = y \odot z$. Thus,

$$F = x' + (y \odot z)$$

3.4.3 * Tabulation Method

K-maps are useful for manually obtaining the minimized standard-form Boolean function for maybe up to, at most, five variables. However, for functions with more than five variables, it becomes very difficult to visualize how the minterms should be combined into subcubes. Moreover, the K-map algorithm is not as straightforward for converting to a computer program. There are **tabulation methods** that are better suited for programming the computer, and thus, can solve any function given in canonical form having any number of variables. One tabulation method is known as the **Quine-McCluskey** method.

Example 3.7: Illustrating the Quime-McCluskey algorithm

We now illustrate the Quine-McCluskey algorithm using the same four-variable function from Example 3.5 and repeated here

$$F(w, x, y, z) = \Sigma(0, 2, 5, 7, 10, 13, 14, 15)$$

To construct the initial table, the minterms are grouped according to the number of 1's in that minterm number's binary representation. For example, m_0 (0000) has no 1's; m_2 (0010) has one 1; m_5 (0101) has two 1's; etc. Thus, the initial table of 0-subcubes (i.e., subcubes having only one minterm) as obtained from the function stated above is

Group	Subcube Minterms	Subcube Value				Subcube Covered
		w	*x*	*y*	*z*	
G_0	m_0	0	0	0	0	✓
G_1	m_2	0	0	1	0	✓
G_2	m_5	0	1	0	1	✓
	m_{10}	1	0	1	0	✓
G_3	m_7	0	1	1	1	✓
	m_{13}	1	1	0	1	✓
	m_{14}	1	1	1	0	✓
G_4	m_{15}	1	1	1	1	✓

The "Subcube Covered" column is filled in from the next step.

In Step 2, we construct a second table by combining those minterms in adjacent groups from the first table that differ in only one bit position, as shown next. For example, m_0 and m_2 differ in only the *y* bit. Therefore, in the second table, we have an entry for the 1-subcube containing the two minterms, m_0 and m_2. A dash (–) is used in the bit position that is different in the two minterms. Since this 1-subcube covers the two individual minterms, m_0 and m_2, we make a note of it by checking these two minterms in the "Subcube Covered" column in the previous table. This process is equivalent to saying that the two minterms, m_0 ($w'x'y'z'$) and m_2 ($w'x'yz'$), can be combined together and are reduced to the one term, $w'x'z'$. The dash under the y column simply means that y can be either a 0 or a 1, and therefore, y can be discarded. Thus, this second table simply lists all of the 1-subcubes. Again, the "Subcube Covered" column in this second table will be filled in from the third step.

Group	Subcube Minterms	Subcube Value				Subcube Covered
		w	x	y	z	
G_0	m_0,m_2	0	0	–	0	
G_1	m_2,m_{10}	–	0	1	0	
G_2	m_5,m_7	0	1	–	1	✓
	m_5,m_{13}	–	1	0	1	✓
	m_{10},m_{14}	1	–	1	0	
G_3	m_7,m_{15}	–	1	1	1	✓
	m_{13},m_{15}	1	1	–	1	✓
	m_{14},m_{15}	1	1	1	–	

In Step 3, we perform the same matching process as before. We look for subcubes in adjacent groups that differ in only one bit position. In the matching, the dash must also match. These subcubes are combined to create the next subcube table. The resulting table, however, is a table containing 2-subcubes. From the above 1-subcube table, we get the following 2-subcube table:

Group	Subcube Minterms	Subcube Value				Subcube Covered
		w	x	y	z	
G_2	m_5,m_7,m_{13},m_{15}	–	1	–	1	

From the 1-subcube table, subcubes m_5m_7 and $m_{13}m_{15}$ can be combined together to form the subcube $m_5m_7m_{13}m_{15}$ in the 2-subcube table, since they differ in only the w bit. Similarly, subcubes m_5m_{13} and m_7m_{15} from the 1-subcube table can also be combined together to form the subcube, $m_5m_7m_{13}m_{15}$, because they differ in only the y bit. From both of these combinations, the resulting subcube is the same. Therefore, we have the four checks in the 1-subcube table, but only one resulting subcube in the 2-subcube table. Notice that in the subcube $m_5m_7m_{13}m_{15}$, there are two dashes; one that is carried over from the Step 2, and one for where the bit is different from the current step.

We continue to repeat the matching step as long as there are adjacent subcubes that differ in only one bit position. We stop when there are no more subcubes that can be combined. The prime implicants are those subcubes that are not covered, (i.e., those without a check mark in the "Subcube Covered" column). The only subcube in the 2-subcube table does not have a check mark, and it has the value $x = 1$ and $z = 1$; thus, we get the prime implicant xz. The 1-subcube table has four subcubes that do not have a check mark; they are the four prime implicants: $w'x'z'$, $x'yz'$, wyz', and wxy. Note that these prime implicants may not be necessarily all in the last table. These five prime implicants (xz, $w'x'z'$, $x'yz'$, wyz', and wxy) are exactly the same as those obtained in Example 3.5. ♦

3.5 * Timing Hazards and Glitches

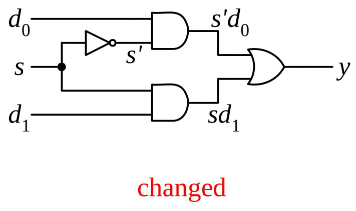
As you probably know, things in practice don't always work according to what you learn in school. Hazards and glitches in circuits are such examples of things that may go awry. In our analysis of combinational circuits, we have been performing only functional analysis. A functional analysis assumes that there is no delay for signals to pass from the input to the output of a gate. In other words, we look at a circuit only with respect to its logical operation as defined by the Boolean theorems. We have not considered the timing of the circuit. When a circuit is actually implemented, the timing of the circuit (that is, the time for the signals to pass from the input of a logic gate to the output) is very critical and must be treated with care. Otherwise, an actual implementation of the circuit may not work according to the functional analysis of the same circuit. **Timing hazards** are problems in a circuit as a result of timing issues. These problems can be observed only from a timing analysis of the circuit or from an actual implementation of the circuit. A functional analysis of the circuit will not reveal timing hazard problems.

A **glitch** is when a signal is expected to be stable (from a functional analysis), but it changes value for a brief moment and then goes back to what it is expected to be. For example, if a signal is expected to be at a stable 0, but instead, it goes up to a 1 and then drops back to a 0 very quickly. This sudden, unexpected transition of the signal is a glitch, and the circuit having this behavior contains a hazard.

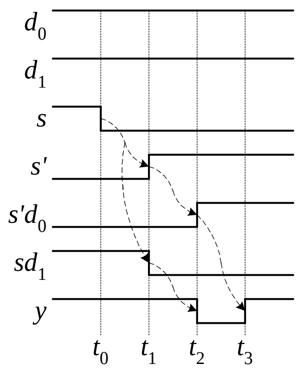
Take, for example, the simple 2-to-1 multiplexer circuit shown in Figure 3.9(a). Let us assume that both d_0 and d_1 are at a constant 1 and that s goes from a 1 to a 0. For a functional analysis of the circuit, the output y should remain at a constant 1. However, if we perform a timing analysis of the circuit, we will see something different in the timing diagram. Let us assume that all of the logic gates in the circuit have a delay of one time unit. The resulting timing trace is shown in Figure 3.9(b). At time t_0, s drops to a 0. Since it takes one time unit for s to be inverted through the inverter, s' changes to a 1 after one time unit at time t_1. At the same time, it takes the bottom AND gate one time unit for the output sd_1 to change to a 0 at time t_1. However, the top AND gate will not see any input change until time t_1, and when it does, it takes another one time unit for its output $s'd_0$ to rise to a 1 at time t_2. Starting at time t_1, both inputs of the OR gate are 0, so after one time unit, the OR gate outputs a 0 at time t_2. At time t_2, when the top AND gate outputs a 1, the OR gate will take this 1 input and outputs a 1 after one time unit at t_3. So between times t_2 and t_3, output y unexpectedly drops to a 0 for one time unit and then rises back to a 1. Hence, the output signal y has a glitch, and the circuit has a hazard.

As you may have noticed, glitches in a signal are caused by multiple sources having paths of different delays driving that signal. These types of simple glitches can be solved easily using K-maps. A glitch generally occurs if, by simply changing one input, we have to go out of one prime implicant in a K-map and into an adjacent one (i.e., moving from one subcube to another). The glitch can be eliminated by adding an extra prime implicant, so that when going from one prime implicant to the adjacent one, we remain inside the third prime implicant.

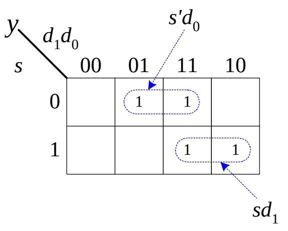
Figure 3.9(c) shows the K-map with the two original prime implicants, $s'd_0$ and sd_1, that correspond to the circuit in Figure 3.9(a). When we change s from a 1 to a 0, we have to go out of the prime implicant sd_1 and into the prime implicant $s'd_0$. Figure 3.9(d) shows the addition of the extra prime implicant d_1d_0. This time, when moving from the prime implicant sd_1 to the prime implicant $s'd_0$, we remain inside the new prime implicant d_1d_0. The 2-to-1 multiplexer circuit with the extra prime implicant d_1d_0 added, as shown in Figure 3.9(e) will prevent the glitch from happening.



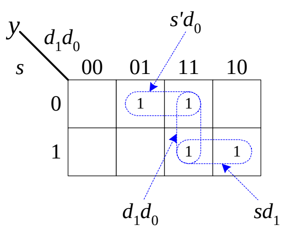
(a)



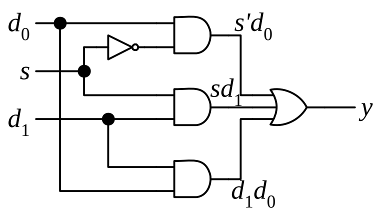
(b)



(c)



(d)



(e)

Figure 3.9 Example of a glitch: (a) 2-to-1 multiplexer circuit with glitches; (b) timing trace; (c) K-map with

glitches; (d) K-map without glitches; (e) 2-to-1 multiplexer circuit without glitches.

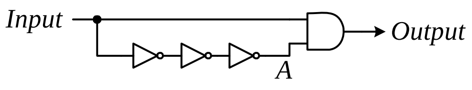
3.5.1 Using Glitches

Sometimes, we can use glitches to our advantage, as shown in the following example.

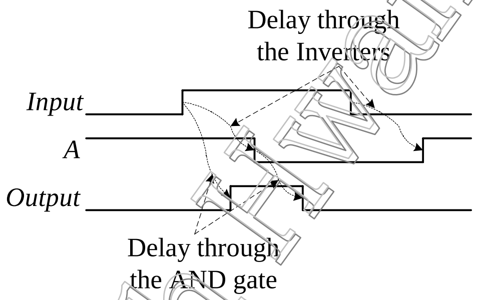
Example 3.8: A one-shot circuit using glitches

A circuit that outputs a single, short pulse when given an input of arbitrary time length is known as a *one-shot*. A one-shot circuit is used, for example, for generating a single, short 1 pulse when a key is pressed. Sometimes, when a key is pressed, we do not want to generate a continuous 1 signal for as long as the key is pressed. Instead, we want the output signal to be just a single, short pulse, even if the key is still being pressed.

Since logic gates have an inherent signal delay, we can use this delay to determine the duration of the short pulse that we want. This short pulse, of course, is really just a glitch in the circuit. Figure 3.10(a) shows a sample one-shot circuit using signal delays through three inverters; Figure 3.10(b) shows a sample timing trace for it.



(a)



(b)

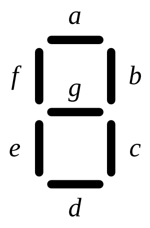
Figure 3.10 A one-shot circuit: (a) using signal delay through three inverters; (b) timing trace.

Initially, assume that the value for *Input* is a 0, and point *A* is a 1; therefore, the output of the AND gate is 0. When we set *Input* to a 1 momentarily, both inputs to the AND gate will be 1's, and so after a delay through the AND gate, *Output* will be a 1. After a delay through the three inverters, with *Input* still at 1, point *A* will go to a 0, and *Output* will change back to a 0. When we set *Input* back to a 0, *Output* will continue to be a 0. After the delay through the inverters when point *A* goes back to a 1, *Output* remains at 0.

As a result, a glitch is created by the signal delay through the three inverters. This glitch, however, is the short 1 pulse that we want, and the length of this pulse is determined by the delay through the inverters. With this one-shot circuit, it does not matter how long the input key is being pressed, the output signal will always be the same 1 pulse each time that the key is pressed. ♦

3.6 BCD to 7-Segment Decoder

We will now synthesize the circuit for a BCD to 7-segment decoder for driving a 7-segment LED display. The decoder converts a 4-bit binary coded decimal (BCD) input to seven output signals for turning on the seven lights in a 7-segment LED display. The 4-bit input encodes the binary representation of a decimal digit. Given the decimal digit input, the seven output lines are turned on in such a way so that the LED displays the corresponding digit. The 7-segment LED display schematic with the names of each segment labeled is shown here.



The operation of the BCD to 7-segment decoder is specified in the truth table in Figure 3.11. The four inputs to the decoder are i_3, i_2, i_1, and i_0, and the seven outputs for each of the seven LEDs are labeled *a*, *b*, *c*, *d*, *e*, *f*, and *g*. For each input combination, the corresponding digit to display on the 7-segment LED is shown in the "Display" column. The segments that need to be turned on for that digit will have a 1, while the segments that need to be turned off for that digit will have a 0. For example, for the 4-bit input 0000, which corresponds to the digit 0, segments *a*, *b*, *c*, *d*, *e*, and *f* need to be turned on, while segment *g* needs to be turned off.

Notice that the input combinations 1010 to 1111 are not used, and so don't-care values are assigned to all of the segments for these six combinations.

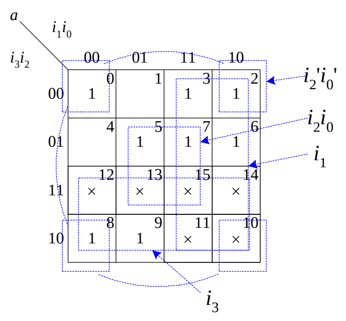
Inputs				Decimal Digit	Display	*a*	*b*	*c*	*d*	*e*	*f*	*g*
i_3	i_2	i_1	i_0									
0	0	0	0	0		1	1	1	1	1	1	0
0	0	0	1	1		0	1	1	0	0	0	0
0	0	1	0	2		1	1	0	1	1	0	1
0	0	1	1	3		1	1	1	1	0	0	1
0	1	0	0	4		0	1	1	0	0	1	1
0	1	0	1	5		1	0	1	1	0	1	1
0	1	1	0	6		1	0	1	1	1	1	1
0	1	1	1	7		1	1	1	0	0	0	0
1	0	0	0	8		1	1	1	1	1	1	1
1	0	0	1	9		1	1	1	0	0	1	1
Rest of the Combinations						×	×	×	×	×	×	×

Figure 3.11 Truth table for the BCD to 7-segment decoder.

From the truth table in Figure 3.11, we are able to specify seven equations that are dependent on the four inputs for each of the seven segments. For example, the canonical form equation for segment *a* is

$$a = i_3'i_2'i_1'i_0' + i_3'i_2'i_1i_0' + i_3'i_2'i_1i_0 + i_3'i_2i_1'i_0 + i_3'i_2i_1i_0' + i_3'i_2i_1i_0 + i_3i_2'i_1'i_0' + i_3i_2'i_1'i_0$$

Before implementing this equation directly in a circuit, we want to simplify it first using the K-map method. The K-map for the equation for segment *a* is



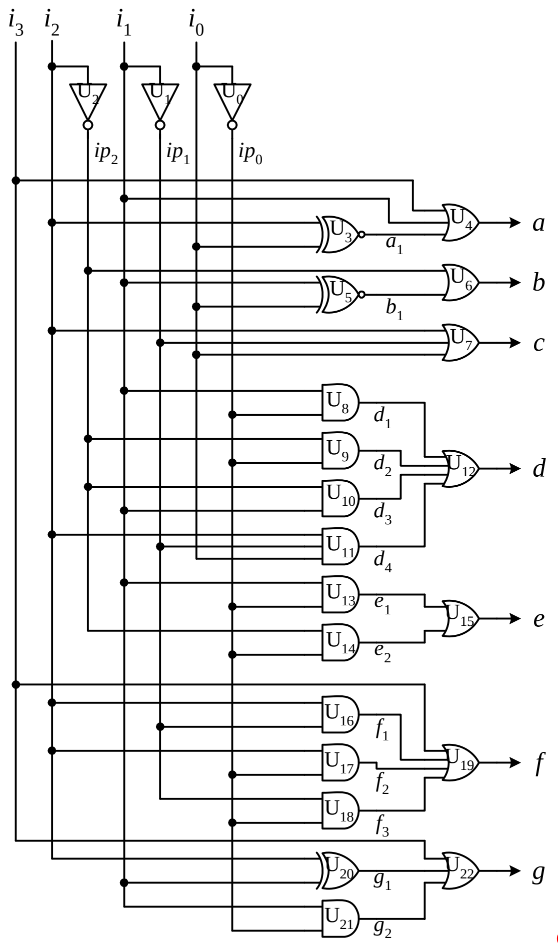
From evaluating the K-map, we derive the simpler equation for segment *a* as

$$a = i_3 + i_1 + i_2'i_0' + i_2i_0 = i_3 + i_1 + (i_2 \odot i_0)$$

Proceeding in a similar manner, we get the following remaining six equations

$$\begin{aligned}
b &= i_2' + (i_1 \odot i_0)\\
c &= i_2 + i_1' + i_0\\
d &= i_1i_0' + i_2'i_0' + i_2'i_1 + i_2i_1'i_0\\
e &= i_1i_0' + i_2'i_0'\\
f &= i_3 + i_2i_1' + i_2i_0' + i_1'i_0'\\
g &= i_3 + (i_2 \oplus i_1) + i_1i_0'
\end{aligned}$$

From these seven simplified equations, we can now implement the circuit, as shown in Figure 3.12. The labeling of the nodes and gates in the drawing will be explained and used in Section 3.7.1.


i_3 i_2 i_1 i_0
U_2 U_1 U_0
ip_2 ip_1 ip_0
U_3 U_4 a_1 a
U_5 U_6 b_1 b
U_7 c
U_8 U_9 U_{10} U_{11} U_{12} d_1 d_2 d_3 d_4 d
U_{13} U_{14} U_{15} e_1 e_2 e
U_{16} U_{17} U_{18} U_{19} f_1 f_2 f_3 f
U_{20} U_{21} U_{22} g_1 g_2 g

check U3 and U5

Figure 3.12 Circuit for the BCD to 7-segment decoder.

3.7 VHDL for Combinational Circuits

Writing VHDL code to describe a digital circuit can be done using any one of three models or levels of abstraction: **structural**, **dataflow**, or **behavioral**. The choice of which model to use usually depends on what is known about the circuit. At the structural level, which is the lowest level, you first have to manually design the circuit. Having drawn the circuit, you use VHDL to specify the components and gates that are needed by the circuit and how they are connected together by following your circuit exactly. Synthesizing a structural VHDL description of a circuit will produce a netlist that is like your original circuit. The advantage of working at the structural level is that you have full control as to what components are used and how they are connected together. The disadvantage, of course, is that you need to manually come up with the circuit, and so the full capabilities of the synthesizer are not utilized. A simple example of a structural VHDL code for a 2-input multiplexer was shown in Figure 1.11.

At the dataflow level, the circuit is defined using built-in VHDL logic operators (such as the AND, OR, and NOT) that are applied to input signals. In order to work at this level, you need to have the Boolean equations for the circuit. Hence, the dataflow level is best suited for describing a circuit that is already expressed as a Boolean function. The equations are easily converted to the required VHDL syntax using signal-assignment statements. A simple example of a dataflow VHDL code for a 2-input multiplexer was shown in Figure 1.9.

All of the statements used in the structural and dataflow levels are executed concurrently, as opposed to statements in a computer program, which usually are executed in a sequential manner. In other words, the ordering of the VHDL statements written in the structural or dataflow level does not matter—the results would be exactly the same.

Describing a circuit at the behavioral level is very similar to writing a computer program. You have all of the standard high-level programming constructs—such as the FOR LOOP, WHILE LOOP, IF THEN ELSE, CASE, and variable assignments. The statements are enclosed in a PROCESS block and are executed sequentially. A simple example of a behavioral VHDL code for a 2-input multiplexer was shown in Figure 1.5.

3.7.1 Structural BCD to 7-Segment Decoder

Figure 3.13 shows the structural VHDL code for the BCD to 7-segment decoder based on the circuit shown in Figure 3.12. The code starts with declaring and defining all of the components needed in the circuit. For this decoder circuit, only basic gates (such as the NOT gate, 2-input AND, 3-input AND, etc.) are used. The ENTITY statement is used to declare all of these components, and the ARCHITECTURE statement is used to define the operation of these components. Since we are using only simple gates, defining these components using the dataflow model is the simplest. For more complex components (as we will see in later chapters), we want to choose the model that is best suited for the information that we have available for the circuit. The reason why the code shown in Figure 3.13 is structural is not because of how these components are defined, but rather on how these components are connected together to form the enclosing entity; in this case, the *bcd* entity. Notice that the LIBRARY and USE statements need to be repeated for every ENTITY declaration.

The actual structural code begins with the *bcd* ENTITY declaration. The *bcd* circuit shown in Figure 3.12 has four input signals: i_3, i_2, i_1, and i_0, and seven output signals: *a*, *b*, *c*, *d*, *e*, *f*, and *g*. These signals are declared in the PORT list using the keyword IN for the input signals, and OUT for the output signals; both of which are of type STD_LOGIC.

The ARCHITECTURE section begins by specifying the components needed in the circuit using the COMPONENT statement. The port list in the COMPONENT statements must match exactly the port list in the entity declarations of the components. They must match not only in the number, direction, and type of the signals but also in the names given to the signals. Note also that names in the component port list can be the same as the names in the *bcd* entity port list, but they are not the same signals. For example, the *and2gate* component port list and the *bcd* entity port list both have two signals called i_1 and i_2. References to these two signals in the body of the *bcd* architecture are for the signals declared in the *bcd* entity.

After the COMPONENT statements, the internal node signals are declared using the SIGNAL statement. The names listed are the same as the internal node names used in the circuit in Figure 3.12 for easy reference.

Following all of the declarations, the body of the architecture starts with the keyword BEGIN. For each gate used in the circuit, there is a corresponding PORT MAP statement. Each PORT MAP statement begins with an optional label (i.e., U_1, U_2, and so on) followed by the name of the component (as previously declared with the COMPONENT statements) to use. Again, the labels used in the PORT MAP statements correspond to the labels on the gates in the circuit in Figure 3.12. The parameter list in the PORT MAP statement matches the port list in the component declaration. For example, U_0 is instantiated with the component *notgate*. The first parameter in the PORT MAP statement is the input signal i_0, and the second parameter is the output signal ip_0. U_4 is instantiated with the 3-input OR gate. The three inputs are i_3, i_1, and a_1, and the output is a. Here, a_1 is the output from the 2-input XNOR gate of U_3. The rest of the PORT MAP statements in the program are obtained in a similar manner.

All of the PORT MAP statements are executed concurrently, and therefore, the ordering of these statements is irrelevant. In other words, changing the ordering of these statements will still produce the same result. Any time when a signal in a PORT MAP statement changes value (i.e., from a 0 to a 1 or vice versa) that PORT MAP statement is executed.

```
----------------- NOT gate -----------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY notgate IS PORT(
  i: IN STD_LOGIC;
  o: OUT STD_LOGIC);
END notgate;
ARCHITECTURE Dataflow OF notgate IS
BEGIN
  o <= NOT i;
END Dataflow;

----------------- 2-input AND gate ---------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY and2gate IS PORT(
  i1, i2: IN STD_LOGIC;
  o: OUT STD_LOGIC);
END and2gate;
ARCHITECTURE Dataflow OF and2gate IS
BEGIN
  o <= i1 AND i2;
END Dataflow;

----------------- 3-input AND gate ---------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY and3gate IS PORT(
  i1, i2, i3: IN STD_LOGIC;
  o: OUT STD_LOGIC);
END and3gate;
ARCHITECTURE Dataflow OF and3gate IS
BEGIN
  o <= (i1 AND i2 AND i3);
END Dataflow;

----------------- 2-input OR gate ----------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY or2gate IS PORT(
  i1, i2: IN STD_LOGIC;
  o: OUT STD_LOGIC);
```

```
END or2gate;
ARCHITECTURE Dataflow OF or2gate IS
BEGIN
  o <= i1 OR i2;
END Dataflow;

----------------- 3-input OR gate ----------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY or3gate IS PORT(
  i1, i2, i3: IN STD_LOGIC;
  o: OUT STD_LOGIC);
END or3gate;
ARCHITECTURE Dataflow OF or3gate IS
BEGIN
  o <= i1 OR i2 OR i3;
END Dataflow;

----------------- 4-input OR gate ----------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY or4gate IS PORT(
  i1, i2, i3, i4: IN STD_LOGIC;
  o: OUT STD_LOGIC);
END or4gate;
ARCHITECTURE Dataflow OF or4gate IS
BEGIN
  o <= i1 OR i2 OR i3 OR i4;
END Dataflow;

----------------- 2-input XOR gate ---------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY xor2gate IS PORT(
  i1, i2: IN STD_LOGIC;
  o: OUT STD_LOGIC);
END xor2gate;
ARCHITECTURE Dataflow OF xor2gate IS
BEGIN
  o <= i1 XOR i2;
END Dataflow;

----------------- 2-input XNOR gate --------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY xnor2gate IS PORT(
  i1, i2: IN STD_LOGIC;
  o: OUT STD_LOGIC);
END xnor2gate;
ARCHITECTURE Dataflow OF xnor2gate IS
BEGIN
  o <= NOT(i1 XOR i2);
END Dataflow;
```

```
----------------- bcd entity ---------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY bcd IS PORT(
  i0, i1, i2, i3: IN STD_LOGIC;
  a, b, c, d, e, f, g: OUT STD_LOGIC);
END bcd;
ARCHITECTURE Structural OF bcd IS
  COMPONENT notgate PORT(
    i: IN STD_LOGIC;
    o: OUT STD_LOGIC);
  END COMPONENT;
  COMPONENT and2gate PORT(
    i1, i2: IN STD_LOGIC;
    o: OUT STD_LOGIC);
  END COMPONENT;
  COMPONENT and3gate PORT(
    i1, i2, i3: IN STD_LOGIC;
    o: OUT STD_LOGIC);
  END COMPONENT;
  COMPONENT or2gate PORT(
    i1, i2: IN STD_LOGIC;
    o: OUT STD_LOGIC);
  END COMPONENT;
  COMPONENT or3gate PORT(
    i1, i2, i3: IN STD_LOGIC;
    o: OUT STD_LOGIC);
  END COMPONENT;
  COMPONENT or4gate PORT(
    i1, i2, i3, i4: IN STD_LOGIC;
    o: OUT STD_LOGIC);
  END COMPONENT;
  COMPONENT xor2gate PORT(
    i1, i2: IN STD_LOGIC;
    o: OUT STD_LOGIC);
  END COMPONENT;
  COMPONENT xnor2gate PORT(
    i1, i2: IN STD_LOGIC;
    o: OUT STD_LOGIC);
  END COMPONENT;

  SIGNAL ip0,ip1,ip2,a1,b1,d1,d2,d3,d4,e1,e2,f1,f2,f3,g1,g2: STD_LOGIC;
BEGIN
  U0: notgate PORT MAP(i0,ip0);
  U1: notgate PORT MAP(i1,ip1);
  U2: notgate PORT MAP(i2,ip2);
  U3: xnor2gate PORT MAP(i2, i0, a1);
  U4: or3gate PORT MAP(i3, i1, a1, a);
  U5: xnor2gate PORT MAP(i1, i0, b1);
  U6: or2gate PORT MAP(ip2, b1, b);
  U7: or3gate PORT MAP(i2, ip1, i0, c);
  U8: and2gate PORT MAP(i1, ip0, d1);
  U9: and2gate PORT MAP(ip2, ip0, d2);
  U10: and2gate PORT MAP(ip2, i1, d3);
  U11: and3gate PORT MAP(i2, ip1, i0, d4);
  U12: or4gate PORT MAP(d1, d2, d3, d4, d);
  U13: and2gate PORT MAP(i1, ip0, e1);
  U14: and2gate PORT MAP(ip2, ip0, e2);
  U15: or2gate PORT MAP(e1, e2, e);
  U16: and2gate PORT MAP(i2, ip1, f1);
  U17: and2gate PORT MAP(i2, ip0, f2);
  U18: and2gate PORT MAP(ip1, ip0, f3);
  U19: or4gate PORT MAP(i3, f1, f2, f3, f);
  U20: xor2gate PORT MAP(i2, i1, g1);
  U21: and2gate PORT MAP(i1, ip0, g2);
  U22: or3gate PORT MAP(i3, g1, g2, g);
END Structural;
```

Figure 3.13 Structural VHDL code of the BCD to 7-segment decoder.

3.7.2 Dataflow BCD to 7-Segment Decoder

Figure 3.14 shows the dataflow VHDL code for the BCD to 7-segment decoder based on the Boolean equations derived in Section 3.6. The ENTITY declaration for this dataflow code is exactly the same as that for the structural code, since the interface for the decoder remains the same.

In the ARCHITECTURE section, seven concurrent signal assignment statements are used: one for each of the seven Boolean equations, which corresponds to the seven LED segments. For example, the equation for segment *a* is

$$a = i_3 + i_1 + (i_2 \odot i_0)$$

This is converted to the signal assignment statement:

a <= i3 OR i1 OR (i2 XNOR i0);

Proceeding in a similar manner, we obtain the signal assignment statements in the dataflow code for the remaining six equations.

All of the signal assignment statements are executed concurrently, and therefore, the ordering of these statements is irrelevant. In other words, changing the ordering of these statements will still produce the same result. Any time when a signal on the right-hand side of an assignment statement changes value (i.e., from a 0 to a 1 or vice versa) that assignment statement is executed.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY bcd IS PORT(
  i0, i1, i2, i3: IN STD_LOGIC;
  a, b, c, d, e, f, g: OUT STD_LOGIC);
END bcd;
ARCHITECTURE Dataflow OF bcd IS
BEGIN
  a <= i3 OR i1 OR (i2 XNOR i0);                                  -- seg a
  b <= (NOT i2) OR NOT (i1 XOR i0);                               -- seg b
  c <= i2 OR (NOT i1) OR i0;                                      -- seg c
  d <= (i1 AND NOT i0) OR (NOT i2 AND NOT i0)                     -- seg d
            OR (NOT i2 AND i1) OR (i2 AND NOT i1 AND i0);
  e <= (i1 AND NOT i0) OR (NOT i2 AND NOT i0);                    -- seg e
  f <= i3 OR (i2 AND NOT i1)                                      -- seg f
            OR (i2 AND NOT i0) OR (NOT i1 AND NOT i0);
  g <= i3 OR (i2 XOR i1) OR (i1 AND NOT i0);                      -- seg g
END Dataflow;
```

Figure 3.14 Dataflow VHDL code of the BCD to 7-segment decoder.

3.7.3 Behavioral BCD to 7-Segment Decoder

The behavioral VHDL code for the BCD to 7-segment decoder is shown in Figure 3.15. The port list for this entity is slightly different from the two entities in the previous sections. Instead of having the four separate input signals, i_0, i_1, i_2, and i_3, we have declared a vector, *I*, of length four. This vector, *I*, is declared with the type keyword STD_LOGIC_VECTOR, that is, a vector of type STD_LOGIC. The length of the vector is specified by the range (3 DOWNTO 0). The first number (3) in the range denotes the index of the most significant bit of the vector, and the second number (0) in the range denotes the index of the least significant bit of the vector. Likewise, the seven output signals, *a* to *g*, are replaced with the STD_LOGIC_VECTOR *Segs* of length 7. This time, however, the keyword TO is used in the range to mean that the most significant bit in the vector is index 1 and the least significant bit in the vector is index 7.

In the architecture section, a PROCESS statement is used. All of the statements inside the process block are executed sequentially. The process block itself, however, is treated as a single concurrent statement. Thus, the architecture section can have two or more process blocks together with other concurrent statements, and these will all execute concurrently.

The parenthesized list of signals after the PROCESS keyword is referred to as the **sensitivity list**. The purpose of the sensitivity list is that, when a value for any of the listed signals changes, the entire process block is executed from the beginning to the end.

In the code, there is a CASE statement inside the process block. Depending on the value of *I*, one of the WHEN parts will be executed. A WHEN part consists of the keyword WHEN followed by a constant value for the variable *I* to match, followed by the symbol "=>." The statement or statements after the symbol "=>" is executed when *I* matches that corresponding constant. In the code, all of the WHEN parts contain one signal assignment statement. All of the signal assignment statements assign a string of seven bits to the output signal *Segs*. This string of seven bits corresponds to the on-off values of the seven segments, *a* to *g*, as shown in the 7-segment decoder truth table of Figure 3.11. For example, looking at the truth table, we see that when *I* = "0000" (that is, for the decimal digit 0) we want all of the segments to be on except for segment *g*. Recall that in the declaration of the *Segs* vector, the most significant bit, which is the leftmost bit in the bit string, is index 1, and the least significant bit, which is the rightmost bit, is index 7. In VHDL, the notation *Segs*(*n*) is used to denote the index *n* of the *Segs* vector. In the code, we have designated *Segs*(1) for segment *a*, *Segs*(2) for segment *b*, and so on to *Segs*(7) for segment *g*. So, in order to display the decimal digit 0, we need to assign the bit string "1111110" to *Segs*.

If the value of *I* does not match any of the WHEN parts, then the WHEN OTHERS part will be chosen. In this case, all of the segments will be turned off. Notice that for both the structural and the dataflow code, the segments are not all turned off when *I* is one of these values. Instead, a certain combination of LEDs are turned on because the K-maps assigned some of the don't-cares to 1's. If we assign all the don't-cares to 0, then all the LEDs will be turned off. An alternative to turning all of the segments off for the remaining six cases is to display the six alphabets, *A*, *b*, *C*, *d*, *E*, and *F*, for the six hexadecimal digits. The two letters, *b*, and *d*, have to be displayed in lower case, because otherwise, it will be the same as the numbers 8 and 0, respectively.

A sample simulation trace of the behavioral 7-segment decoder code is shown in Figure 3.16.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY bcd IS PORT (
  I: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
  Segs: OUT STD_LOGIC_VECTOR (1 TO 7));
END bcd;

ARCHITECTURE Behavioral OF bcd IS
BEGIN
  PROCESS(I)
  BEGIN
    CASE I IS
    WHEN "0000" => Segs <= "1111110";
    WHEN "0001" => Segs <= "0110000";
    WHEN "0010" => Segs <= "1101101";
    WHEN "0011" => Segs <= "1111001";
    WHEN "0100" => Segs <= "0110011";
    WHEN "0101" => Segs <= "1011011";
    WHEN "0110" => Segs <= "1011111";
    WHEN "0111" => Segs <= "1110000";
    WHEN "1000" => Segs <= "1111111";
    WHEN "1001" => Segs <= "1110011";
    WHEN OTHERS => Segs <= "0000000";
    END CASE;
  END PROCESS;
END Behavioral;
```

Figure 3.15 Behavioral VHDL code of the BCD to 7-segment decoder.

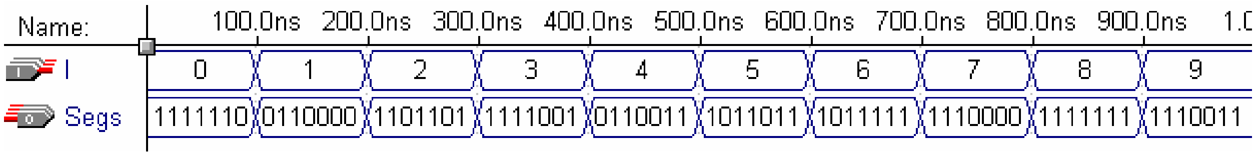


Figure 3.16 A sample simulation trace of the behavioral 7-segment decoder code.

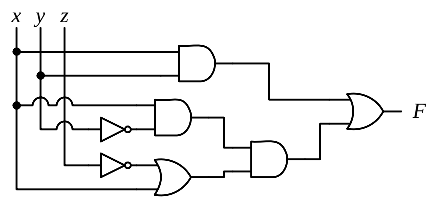
3.8 Summary Checklist

- ❑ Combinational circuit
- ❑ Analysis of combinational circuit
- ❑ Synthesis of combinational circuit
- ❑ Technology mapping
- ❑ Using K-maps to minimize a Boolean function
- ❑ The use of “don’t-cares”
- ❑ Using “don’t-cares” in a K-map
- ❑ Using the Quine-McCluskey method to minimize a Boolean function
- ❑ Timing hazards and glitches
- ❑ How to eliminate simple glitches
- ❑ Writing structural, dataflow, and behavioral VHDL code
- ❑ Be able to analyze any combinational circuit by deriving its truth table or Boolean function
- ❑ Be able to synthesize a combinational circuit from a truth table or Boolean function
- ❑ Be able to reduce any combinational circuit to its smallest size

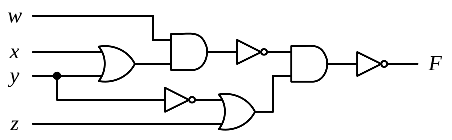
3.9 Problems

3.1 Derive the truth table for the following circuits:

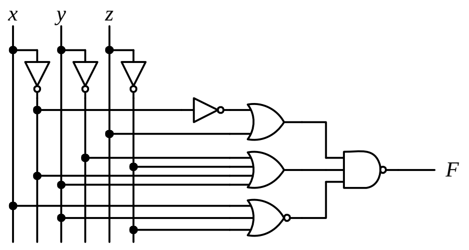
a)



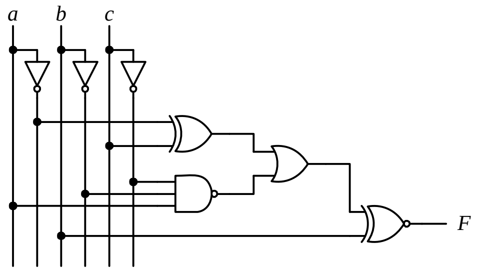
b)



c)



d)



3.2 Derive the Boolean function directly from the circuits in Problem 3.1.

3.3 Draw the circuit diagram that implements the following truth tables.

a)

a	b	c	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

b)

w	x	y	z	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

c)

w	x	y	z	F_1	F_2
0	0	0	0	1	1
0	0	0	1	0	1
0	0	1	0	0	1
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	1	1	1
0	1	1	0	1	0
0	1	1	1	0	0
1	0	0	0	0	1
1	0	0	1	1	1
1	0	1	0	1	0
1	0	1	1	0	0
1	1	0	0	1	1
1	1	0	1	0	1
1	1	1	0	0	1
1	1	1	1	1	1

d)

N_3	N_2	N_1	N_0	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

3.4 Draw the circuit diagram that implements the following expressions:

a) $F(x, y, z) = \Sigma(0, 1, 6)$
b) $F(w,x, y, z) = \Sigma(0, 1, 6)$
c) $F(w,x, y, z) = \Sigma(2, 6, 10, 11, 14, 15)$
d) $F(x, y, z) = \Pi(0, 1, 6)$
e) $F(w,x, y, z) = \Pi(0, 1, 6)$
f) $F(w,x, y, z) = \Pi(2, 6, 10, 11, 14, 15)$

3.5 Draw the circuit diagram that implements the following Boolean functions using as few basic gates as possible, but without modifying the equation.

a) $F = xy' + x'y'z + xyz'$
b) $F = w'z' + w'xy + wx'z + wxyz$
c) $F = w'xy'z + w'xyz + wxy'z + wxyz$
d) $F = N_3'N_2'N_1N_0' + N_3'N_2'N_1N_0 + N_3N_2'N_1N_0' + N_3N_2'N_1N_0 + N_3N_2N_1'N_0' + N_3N_2N_1N_0$

e) $F = [(x \odot y)' + (xyz)'] (w' + x + z)$
f) $F = x \oplus y \oplus z$
g) $F = [w'xy'z + w'z (y \oplus x)]'$

3.6 Draw the circuit diagram that implements the Boolean functions in Problem 3.5 using only 2-input AND, 2-input OR, and NOT gates.

3.7 Design a circuit that inputs a 4-bit number. The circuit outputs a 1 if the input number is any one of the following numbers: 2, 3, 10, 11, 12, and 15. Otherwise, it outputs a 0.

3.8 Design a circuit that inputs a 4-bit number. The circuit outputs a 1 if the input number is greater than or equal to 5. Otherwise, it outputs a 0.

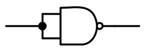
3.9 Design a circuit that inputs a 4-bit number. The circuit outputs a 1 if the input number has an even number of zeros. Otherwise, it outputs a 0.

3.10 Construct the following circuit. The circuit has five input signals and one output signal. The five input lines are labeled W, X, Y, Z, and E, and the output line is labeled F. E is used to enable (turn on) or disable (turn off) the circuit; thus, when E = 0, the circuit is disabled, and F is always 0. When E = 1, the circuit is enabled, and F is determined by the value of the four input signals, W, X, *Y*, and Z, where W is the most significant bit. If the value is odd, then F = 1, otherwise F = 0.

3.11 Draw the smallest circuit that inputs two 2-bit numbers. The circuit outputs a 2-bit number that represents the count of the number of even numbers in the inputs. The number 0 is taken as an even number. For example, if the two input numbers are 0 and 3, then the circuit outputs the number 1 in binary. If the two input numbers are 0 and 2, then the circuit outputs the number 2 in binary. Show your work by deriving the truth table, the equation, and finally the circuit. You need to minimize all of the equations to standard forms.

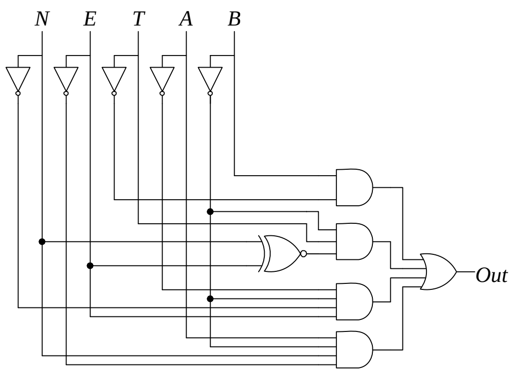
3.12 Derive and draw the circuit that inputs two 2-bit unsigned numbers. The circuit outputs a 3-bit signed number that represents the difference between the two input numbers (i.e., it is the result of the first number minus the second number). Derive the truth table and equations in canonical form.

3.13 Use Boolean algebra to show that the following circuit is equivalent to the NOT gate.



3.14 Construct a 4-input NAND gate circuit using only 2-input NAND gates.

3.15 Implement the following circuit using as few NAND gates (with any number of inputs) as possible.

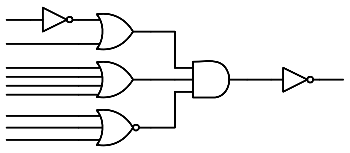


3.16 Draw the circuit diagram that implements the Boolean functions in Problem 3.5 using only 2-input NAND gates.

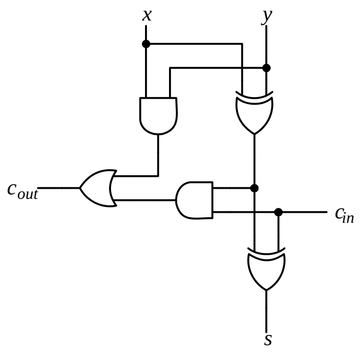
3.17 Draw the circuit diagram that implements the Boolean functions in Problem 3.5 using only 3-input NAND gates.

3.18 Draw the circuit diagram that implements the Boolean functions in Problem 3.5 using only 3-input NOR gates.

3.19 Convert the following circuit as is (i.e., do not reduce it first) to use only 2-input NOR gates.



3.20 Convert the following full adder circuit to use only eleven 2-input NAND gates.



3.21 Perform a timing analysis of the circuit shown in Figure 3.9(e) to see that the circuit does not produce any glitches.

3.22 Derive a circuit for the 2-input XOR gate that uses only 2-input NAND gates.

3.23 Use K-maps to reduce the Boolean functions represented by the truth tables in Problem 3.3 to standard form.

3.24 Use K-maps to reduce the Boolean functions in Problem 3.4 to standard form.

3.25 Use K-maps to reduce the Boolean functions in Problem 3.5 to standard form.

3.26 List all of the PIs, EPIs, and all of the minimized standard form solutions for the following equation.

$$F(v,w,x,y,z) = \Pi(2,3,4,5,6,7,8,9,11,13,15,18,19,20,21,22,29,30,31)$$

3.27 Use K-maps to reduce the following 4-variable Boolean functions $F(w, x, y, z)$ to standard form:

a) 1-minterms: m_2, m_3, m_4, m_5
Don't-care minterms: m_{10}, m_{11}, m_{12}, m_{13}, m_{14}, m_{15}
b) 1-minterms: 1, 3, 4, 7, 9
Don't-care minterms: 0, 2, 13, 14, 15
c) 1-minterms: 2, 3, 8, 9
Don't-care minterms: 1, 5, 6, 7, 13, 15

3.28 Use K-maps to reduce the following 5-variable Boolean functions $F(v, w, x, y, z)$ to standard form:

a) 1-minterms: 1, 3, 4, 7, 9
Don't-care minterms: 0, 2, 13, 14, 15
b) 1-minterms: 2, 4, 10, 15, 16, 21, 26, 29
Don't-care minterms: 5, 7, 13, 18, 23, 24, 31

3.29 Use the Quine-McCluskey method to simplify the function $f(w,x,y,z) = \Sigma(0,2,5,7,13,15)$. List all the PI's, EPI's, cover lists, and solutions.

3.30 Use the Quine-McCluskey method to reduce the Boolean functions in Problem 3.4 to standard form.

3.31 Write the function that eliminates the static hazard(s) in the function $F = w'z + xyz' + wx'y$.

3.32 Write the function that eliminates the static hazard(s) in the function $F = y'z' + wz + w'x'y$.

3.33 Write the complete structural VHDL code for the Boolean functions in Problem 3.4.

3.34 Write the complete dataflow VHDL code for the Boolean functions in Problem 3.4.

3.35 Write the complete behavioral VHDL code for the Boolean functions in Problem 3.4.

3.36 Write the complete dataflow VHDL code for the Boolean functions in Problem 3.5.

3.37 Write the behavioral VHDL code for converting an 8-bit unsigned binary number to two 4-bit BCD numbers. These two BCD numbers represent the tenth and unit digits of a decimal number. Also, turn on the decimal point LED for the unit digit if the 8-bit binary number is in the one hundreds range, and turn on the decimal point LED for the tenth digit if the 8-bit binary number is in the two hundreds range. This circuit is used as the output circuit for many designs in later chapters.

Index