regardless of the value of *x*. Thus, the grouping coalesces two minterms into one product term by eliminating *x*.

From the last grouping, we know our final simplified function will have a *y* term. Let's do another grouping to find the next term. First, we'll simplify the equation algebraically. Returning to the original equation for *F*(*x*, *y*), we can use idempotency to duplicate one of the minterms:

$$F(x,y)=(x \wedge \neg y) \vee (\neg x \wedge y) \vee (x \wedge y) \vee (x \wedge y)$$

Now we'll do some algebraic manipulation on the first product term and the one we just added:

$$\begin{aligned}(x \wedge \neg y) \vee (x \wedge y) &= (\neg y \vee y) \wedge x \\ &= x\end{aligned}$$

Instead of using algebraic manipulations, we can do this directly on our Karnaugh map, as shown in Figure 4-10. This map shows that separate groups can include the same cell (minterm).

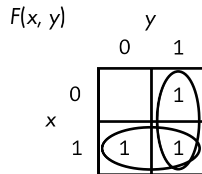

Figure 4-10: A Karnaugh map grouping showing that (x ∧ ¬y) ∨ (¬x ∧ y) ∨ (x ∧ y) = (x ∨ y)

The group in the bottom row represents the product term *x*, and the one in the right-hand column represents *y*, giving us the following minimization:

$$F(x,y)=x \vee y$$

Note that the cell that is included in both groupings, (*x* ∧ *y*), is the term that we duplicated using the idempotent rule in our algebraic solution previously. You can think of including a cell in more than one group as adding a duplicate copy of the cell, like using the idempotent rule in our algebraic manipulation earlier, and then coalescing it with the other cell(s) in the group, thus removing it.

The adjacency rule is automatically satisfied when there are only two variables in the function. But when we add another variable, we need to think about how to order the cells of a Karnaugh map such that we can use the adjacency rule to simplify Boolean expressions.

Karnaugh Map Cell Order

One of the problems with both the binary and BCD codes is that the difference between two adjacent values often involves more than one bit being

changed. In 1943 Frank Gray introduced a code, the *Gray code*, in which adjacent values differ by only one bit. The Gray code was invented because the switching technology of that time was more prone to errors. If one bit was in error, the value represented by a group of bits was off by only one in the Gray code. That's seldom a problem these days, but this property shows us how to order the cells in a Karnaugh map.

Constructing the Gray code is quite easy. Start with one bit:

Decimal	Gray Code
0	0
1	1

To add a bit, first write the mirror image of the existing pattern:

Gray Code
0
1
1
0

Then add a `0` to the beginning of each of the original bit patterns and add a `1` to the beginning of each of the mirror image set to give the Gray code for two bits, as shown in Table 4-9.

Table 4-9: Gray Code for Two Bits

Decimal	Gray Code
0	00
1	01
2	11
3	10

This is the reason the Gray code is sometimes called *reflected binary code* (*RBC*). Table 4-10 shows the Gray code for four bits.

Table 4-10: Gray Code for Four Bits

Decimal	Gray Code	Binary
0	0000	0000
1	0001	0001
2	0011	0010

(continued)

Decimal	Gray Code	Binary
3	0010	0011
4	0110	0100
5	0111	0101
6	0101	0110
7	0100	0111
8	1100	1000
9	1101	1001
10	1111	1010
11	1110	1011
12	1010	1100
13	1011	1101
14	1001	1110
15	1000	1111

Let's compare the binary codes with the Gray codes for the decimal values 7 and 8 in Table 4-10. The binary codes for 7 and 8 are `0111` and `1000`, respectively; all four bits change when stepping only 1 in decimal value. But comparing the Gray codes for 7 and 8, `0100` and `1100`, respectively, only one bit changes, thus satisfying the adjacency rule for a Karnaugh map.

Notice that the pattern of changing only one bit between adjacent values also holds when the bit pattern wraps around. That is, only one bit is changed when going from the highest value (15 for four bits) to the lowest (0).

Karnaugh Map for Three Variables

To see how the adjacency property is important, let's consider a more complicated function. We'll use a Karnaugh map to simplify our function for carry, which has three variables. Adding another variable means that we need to double the number of cells to hold the minterms. To keep the map two-dimensional, we add the new variable to an existing variable on one side of the map. We need a total of eight cells (2^3), so we'll draw it four cells wide and two high. We'll add *z* to the *y*-axis and draw our Karnaugh map with *y* and *z* on the horizontal axis, and *x* on the vertical, as shown in Figure 4-11.

$F(x, y, z)$

x \ yz	00	01	11	10
0	m_0	m_1	m_3	m_2
1	m_4	m_5	m_7	m_6

Figure 4-11: Mapping of three-variable minterms on a Karnaugh map

The order of the bit patterns along the top of the three-variable Karnaugh map is 00, 01, 11, 10, as opposed to 00, 01, 10, 11, which is the Gray code order in Table 4-9. The adjacency rule also holds when wrapping around the edges of the Karnaugh map—that is, going from m_2 to m_0 or going from m_6 to m_4—which means that groups can wrap around the edges of the map. (Other axis labeling schemes will also work, as you'll see when it's Your Turn at the end of this section.)

You saw earlier in this chapter that carry can be expressed as the sum of four minterms:

$$\begin{aligned} C_{out}(c_{in},x,y) &= (\neg c_{in} \wedge x \wedge y) \vee (c_{in} \wedge \neg x \wedge y) \vee (c_{in} \wedge x \wedge \neg y) \vee (c_{in} \wedge x \wedge y) \\ &= m_3 \vee m_5 \vee m_6 \vee m_7 \\ &= \sum (3,5,6,7) \end{aligned}$$

Figure 4-12 shows these four minterms on the Karnaugh map.

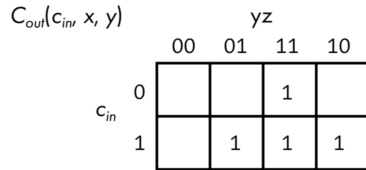

Figure 4-12: Karnaugh map of the function for carry

We look for adjacent cells that can be grouped together to eliminate one variable from the product term. As noted, the groups can overlap, giving the three groups shown in Figure 4-13.

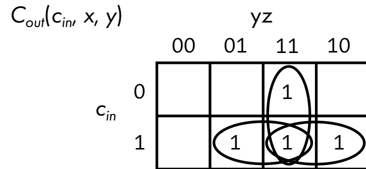

Figure 4-13: A minimum sum of products of the function for carry = 1

Using the three groups in the Karnaugh map in Figure 4-13, we end up with the same equation we got through algebraic manipulations:

$$C_{out}(c_{in},x,y) = (x \wedge y) \vee (c_{in} \wedge x) \vee (c_{in} \wedge y)$$

Simplifying Products of Sums Using Karnaugh Maps

It's equally valid to work with a function that shows when the complement of carry is 0. We did that using maxterms:

$$\begin{aligned} \neg C_{max}(c_{in},x,y) &= (c_{in} \vee \neg x \vee \neg y) \wedge (\neg c_{in} \vee x \vee \neg y) \wedge (\neg c_{in} \vee \neg x \vee y) \wedge (\neg c_{in} \vee \neg x \vee \neg y) \\ &= M_7 \wedge M_6 \wedge M_5 \wedge M_3 \\ &= \prod (3,5,6,7) \end{aligned}$$

Introduction to Computer Organization (Sample Chapter) © 7/24/21 by Robert Plantz

Figure 4-14 shows the arrangement of maxterms on a three-variable Karnaugh map.

$\neg F(x, y, z)$

x \ yz	00	01	11	10
0	M_0	M_1	M_3	M_2
1	M_4	M_5	M_7	M_6

Figure 4-14: Mapping of three-variable maxterms on a Karnaugh map

When working with a maxterm statement of the solution, you mark the cells that evaluate to `0`. The minimization process is the same as when working with minterms, except that you group the cells with `0`s in them.

Figure 4-15 shows a minimization of $\neg C_{out}(c_{in}, x, y)$, the complement of carry.

$\neg C_{out}(c_{in}, x, y)$

c_{in} \ yz	00	01	11	10
0			0	
1		0	0	0

Figure 4-15: A minimum product of sums of the function for NOT carry = `0`

The Karnaugh map in Figure 4-15 leads to the same product of sums we got algebraically for the complement of carry = `0`:

$$\neg C_{out}(c_{in}, x, y) = (\neg x \vee \neg y) \wedge (\neg c_{in} \vee \neg x) \wedge (\neg c_{in} \vee \neg y)$$

If you compare Figures 4-13 and 4-15, you can see a graphic view of De Morgan's law. When making this comparison, keep in mind that Figure 4-13 shows the product terms that get added, and Figure 4-15 shows the sum terms that get multiplied, and the result is complemented. Thus, we exchange `0` and `1` and exchange AND and OR to go from one Karnaugh map to the other.

To further emphasize the duality of minterm and maxterm, compare Figure 4-16(a) and Figure 4-16(b).

$F(x, y, z)$

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

(a)

$\neg F(x, y, z)$

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

(b)

Figure 4-16: Comparison of (a) one minterm and (b) one maxterm

Figure 4-16(a) shows the function:

$$F(x,y,z)=\neg x \wedge \neg y \wedge \neg z$$

Although it's not necessary and usually not done, we have placed a `0` in each of the cells representing minterms not included in this function.

Similarly, in Figure 4-16(b), we have placed a `0` for the maxterm and a `1` in each of the cells representing the maxterms that are not included in the function:

$$\neg F=x \vee y \vee z$$

This comparison graphically shows how a minterm specifies the minimum number of `1`s in a Karnaugh map, while a maxterm specifies the maximum number of `1`s.

Larger Groupings on a Karnaugh Map

Thus far, we have grouped only two cells together on our Karnaugh maps. Let's look at an example of larger groups. Consider a function that outputs `1` when a three-bit number is even. Table 4-11 shows the truth table. It uses `1` to indicate that the number is even and uses `0` to indicate odd.

Table 4-11: Even Values of an Eight-Bit Number

Minterm	*X*	*y*	*z*	Number	*Even*(*x*, *y*, *z*)
m_0	0	0	0	0	1
m_1	0	0	1	1	0
m_2	0	1	0	2	1
m_3	0	1	1	3	0
m_4	1	0	0	4	1
m_5	1	0	1	5	0
m_6	1	1	0	6	1
m_7	1	1	1	7	0

The canonical sum of products for this function is

$$Even(x,y,z)=\sum(0,2,4,6)$$

Figure 4-17 shows these minterms on a Karnaugh map with these four terms grouped together. You can group all four together because they all have adjacent edges.

From the Karnaugh map in Figure 4-17, we can write the equation for showing when a three-bit number is even:

$$Even(x,y,z)=\neg z$$

Even(*x*, *y*, *z*)

x \ *yz*	00	01	11	10
0	1			1
1	1			1

Figure 4-17: Karnaugh map showing even values of three-bit number

The Karnaugh map shows that it does not matter what the values of x and y are, only that $z = 0$.

Adding More Variables to a Karnaugh Map

Each time you add another variable to a Karnaugh map, you need to double the number of cells. The only requirement for the Karnaugh map to work is that you arrange the minterms, or maxterms, according to the adjacency rule. Figure 4-18 shows a four-variable Karnaugh map for minterms. The y and z variables are on the horizontal axis, and w and x are on the vertical.

F(*w*, *x*, *y*, *z*)

wx \ *yz*	00	01	11	10
00	m_0	m_1	m_3	m_2
01	m_4	m_5	m_7	m_6
11	m_{12}	m_{13}	m_{15}	m_{14}
10	m_8	m_9	m_{11}	m_{10}

Figure 4-18: Mapping of four-variable minterms on a Karnaugh map

So far we have assumed that every minterm (or maxterm) is accounted for in our functions. But design does not take place in a vacuum. We might have knowledge about other components of the overall design telling us that some combinations of variable values can never occur. Next, we'll see how to take this knowledge into account in your function simplification process. The Karnaugh map provides an especially clear way to visualize the situation.

Don't Care Cells

Sometimes, you have information about the values that the variables can have. If you know which combinations of values will never occur, then the minterms (or maxterms) that represent those combination are irrelevant. For example, you may want a function that indicates whether one of two possible events has occurred or not, but you know that the two events cannot occur simultaneously. Let's name the events x and y, and let `0` indicate

that the event has not occurred and 1 indicate that it has. Table 4-12 shows the truth table for our function, $F(x, y)$.

Table 4-12: Truth Table for *x* or *y* Occurring, but not Both

x	*y*	*F(x, y)*
0	0	0
0	1	1
1	0	1
1	1	X

We can show that both events cannot occur simultaneously by placing an × in that row. We can draw a Karnaugh map with an × for the minterm that can't exist in the system, as shown in Figure 4-19. The × represents a *don't care* cell—we don't care whether this cell is grouped with other cells or not.

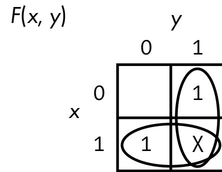

Figure 4-19: Karnaugh map for F(x, y)*, showing a "don't care" cell*

Since the cell that represents the minterm $(x \wedge y)$ is a "don't care" cell, we can include it, or not, in our minimization groupings, leading to the two groupings shown. The Karnaugh map in Figure 4-19 leads us to the solution:

$$F(x,y)=x \vee y$$

which is a simple OR gate. You probably guessed this solution without having to use a Karnaugh map. You'll see a more interesting use of "don't care" cells when you learn about the design of two digital logic circuits at the end of Chapter 7.

Combining Basic Boolean Operators

As mentioned earlier in this chapter, we can combine basic Boolean operators to implement more complex Boolean operators. Now that you know how to work with Boolean functions, we'll design one of the more common operators, the *exclusive or*, often called *XOR*, using the three basic operators, AND, OR, and NOT. It's so commonly used that it has its own circuit symbol.

XOR

The XOR is a binary operator. The result is 1 if one, and only one, of the two operands is 1; otherwise, the result is 0. We'll use $\veebar$ to designate the XOR operation. It's also common to use the $\oplus$ symbol. Figure 4-20 shows the XOR gate operation with inputs x and y.

x	**y**	**x ⊻ y**
0	0	0
0	1	1
1	0	1
1	1	0

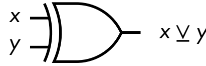

Figure 4-20: The XOR gate acting on two variables, x *and* y

The minterm implementation of this operation is

$$x \veebar y = (x \wedge \neg y) \vee (\neg x \wedge y)$$

The XOR operator can be implemented with two AND gates, two NOT gates, and one OR gate, as shown in Figure 4-21.

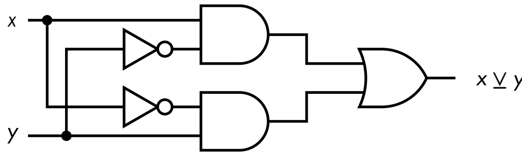

Figure 4-21: XOR gate made from AND, OR, and NOT gates

We can, of course, design many more Boolean operators. But we're going to move on in the next few chapters and see how these operators can be implemented in hardware. It's all done with simple on/off switches.

YOUR TURN

Design a function that will detect all the four-bit integers that are even.

Find a minimal sum of products expression for this function:

$$F(x,y,z) = (\neg x \wedge \neg y \wedge \neg z) \vee (\neg x \wedge \neg y \wedge z) \vee (\neg x \wedge y \wedge \neg z) \vee (x \wedge \neg y \wedge \neg z) \vee (x \wedge y \wedge \neg z) \vee (x \wedge y \wedge z)$$

Find a minimal product of sums expression for this function:

$$F(x,y,z) = (x \vee y \vee z) \wedge (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$$

The arrangement of the variables for a Karnaugh map is arbitrary, but the minterms (or maxterms) need to be consistent with the labeling. Show where each minterm is located with this Karnaugh map axis labeling using the notation of Figure 4-11.

F(*x*, *y*, *z*) — *xy*

z \ *xy*	00	01	11	10
0				
1				

The arrangement of the variables for a Karnaugh map is arbitrary, but the minterms (or maxterms) need to be consistent with the labeling. Show where each minterm is located with this Karnaugh map axis labeling using the notation of Figure 4-11.

F(*x*, *y*, *z*) — *xz*

y \ *xz*	00	01	11	10
0				
1				

Create a Karnaugh map for five variables. You'll probably need to review the Gray code in Table 4-10 and increase it to five bits.

Design a logic function that detects the single-digit prime numbers. Assume that the numbers are coded in four-bit BCD (see Table 2-7 in Chapter 2. The function is `1` for each prime number.

What You've Learned

Boolean operators The basic Boolean operators are AND, OR, and NOT.

Rules of Boolean algebra Boolean algebra provides a mathematical way to work with the rules of logic. AND works like multiplication and OR is similar to addition in elementary algebra.

Simplifying Boolean algebra expressions Boolean functions specify the functionality of a computer. Simplifying these functions leads to a simpler hardware implementation.

Karnaugh maps These provide a graphical way to simplify Boolean expressions.

Introduction to Computer Organization (Sample Chapter) © 7/24/21 by Robert Plantz

Gray code This shows how to order the cells in a Karnaugh map.

Combining basic Boolean operators XOR can be created from AND, OR, and NOT.

The next chapter starts with an introduction to basic electronics that will provide a basis for understanding how transistors can be used to implement switches. From there, we'll look at how transistor switches are used to implement logic gates.