5
Boolean algebra

In this chapter, we look in some detail at Boolean algebra, which we introduced in the previous chapter in the context of creating Boolean expressions and functions. Boolean algebra is important theoretical knowledge; it is a tool used throughout computer science. It is also very important in our discussion of digital logic circuits in the next chapter, since these circuits implement Boolean expressions.

We will focus primarily on using Boolean algebra to prove the functional equivalence of two circuits, which is important for simplifying circuits or otherwise creating circuits that are implemented differently, yet do the same thing. We show functional equivalence by showing that the Boolean algebra representation of one circuit can be derived from the other or vice versa.

Functional equivalence

Two circuits are *functionally equivalent* if, in every case, they produce the same output given the same inputs. The circuits may use different types of gates, different numbers of gates, or have different connections between the gates. However, whenever their inputs are the same, their outputs must be the same as well.

Functional equivalence is important, since one circuit may be superior to another in some way; if we can prove that the two circuits are functionally equivalent, then we are free to use the superior implementation.

One circuit can be better than another, equivalent circuit for several reasons. For example, a circuit might be improved by reducing the number of gates necessary to implement its functionality, since gates have associated costs. These costs include their monetary cost, the amount of a chip's area occupied by the gates, and the time required for the gates to do their work. Fewer gates can result in a cheaper chip, or a chip that has room for more gates on it that can then be used for other functions. Fewer gates in a circuit can also re-

sult in a faster circuit, since each gate takes some small, finite amount of time (usually in the nanosecond, or billionth of a second, range) to compute its output from its inputs. Since gates are electronic circuits themselves, they consume power and produce heat. Fewer gates, then, results in more efficient, cooler chips.

We may also want to find an equivalent circuit for a function that uses only particular kinds of gates. For example, if we have described the function using `AND`, `OR`, and `NOT` gates, but we have available only `NAND` gates. Or we may want to use only `NAND` gates, since they can be simpler, cheaper gates. In either case, we will need to find an equivalent circuit composed only of `NAND` gates.

We show that two logic circuits are functionally equivalent by showing that the Boolean algebra expressions representing them are equivalent. There are in general two ways to prove that two Boolean functions, A and B are equivalent: truth tables and using the laws of Boolean algebra (*algebraic equivalence*).

Proving equivalence using truth tables

To prove equivalence using truth tables, a truth table is created that has *two* output columns, one for each of the expressions in question. If the output columns are the same, then the two expressions are equivalent.

This makes sense, since a truth table lists the output of a function for every possible combination of inputs; so if the outputs are the same for each set of inputs for the two functions, then they are equivalent.

Boolean algebra expressions in which we are interested typically have more than one operator. To facilitate both the construction of the table and also to help convince a reader that our table (and thus our proof) is accurate, we usually include in the table one or more intermediate columns that correspond to portions of the actual functions.

1	2	3	4	5	6	7	8
A	B	$A \oplus B$	$\overline{A}$	$\overline{B}$	$\overline{A}B$	$A\overline{B}$	$\overline{A}B + A\overline{B}$
0	0	0	1	1	0	0	0
0	1	1	1	0	1	0	1
1	0	1	0	1	0	1	1
1	1	0	0	0	0	0	0

Figure 5.1: A truth table-based proof that $A \oplus B \equiv \overline{A}B + A\overline{B}$.

For example, let's see how we would prove that $A \oplus B \equiv \overline{A}B + A\overline{B}$. The truth table in Figure 5.1 shows a truth table-based proof. The columns are numbered only for explanatory purposes. The first two columns are, as usual, for the input variables. Since there are two

variables, there will be four rows. In column 3, we show the values for $A \oplus B$ for the input variable values. Columns 4 and 5 give the values for two subexpressions, applying the `NOT` operator to A and B, respectively. Columns 6 and 7 show larger subexpressions that occur in the results, in this case the two things that are `OR`'ed together (called *disjuncts*, since `OR` is also known as *disjunction*. The final column shows values of the thing we are trying to prove equivalence to. At each step, we are building on what has gone before. We, and anyone else who looks at the truth table, can tell now that the two expressions are the same; it will not need any further explanation.

Proving equivalence using Boolean algebra

Proving equivalence using truth tables is straightforward and easy—why would we want to use anything else?

There are two reasons. First, in some cases, we are given a function, but we don't know what we want to show it is equivalent *to*. This happens, for example, when our task is to minimize a circuit; we can easily create the Boolean function representing the circuit, but we do not know what the minimized circuit and its Boolean function are—otherwise, we would be done. In this case, we can't use a truth table.

The second reason has to do with the size of truth tables, which can get very large, very fast, as we discussed in the preceding chapter. For 10 input variables, the table will have 1024 rows. For 20, 1,048,576—rather inconvenient to write down. In fact, assuming 60 lines per page, this truth table would take up almost 90 200-page books!

So we need something else that will not require so much space and that ideally can be used even when we don't know what one of the functions is, so that we can find a new, minimal function for one we know. *Algebraic substitution* fits the bill. Using this, we can prove that one function is equivalent to another without knowing in advance with the other is and without having to enumerate each possible value of both functions. As an added benefit, proof by algebraic substitution is amenable to automation, using artificial intelligence (AI) programs called *automated theorem provers*.

However, using algebraic equivalence can be a little more challenging than using truth tables. It involves repeatedly substituting one expression for another, equivalent one in a function until we find what we are looking for. We ensure that these substitutions are equivalent by following the *laws* of Boolean algebra or by making substitutions that have been proven to be equivalent before.

What is tricky is picking the right law or previous substitution to

use. We may need to try several things and go down several paths that lead to dead ends before we find a way to create the proof. If we are only making legitimate substitutions, we cannot do anything wrong. However, we can do things that are unhelpful or inelegant. These false starts should not be included in your final proof. (So expect to rewrite your proofs before turning in homework!)

Laws of Boolean algebra

Each step in a proof will need to be justified by a law of Boolean algebra. Most of the laws have a form to be used with `AND` and a form to be used with `OR`. Many of the laws may seem intuitively obvious to you, or they may just make a lot of sense when you think about them. Either can be helpful, because when you feel comfortable with the laws, it will be easier for you to remember and use them.

In this section, we present each law along with a brief discussion that is meant to explain the law (and help to remember it better). However, just as with truth tables for operators, you should not rely on the discussion and your intuition about the laws. The law itself is what you must use in the proof; everything else is a tool to help you work with the law. At the end of the section, we have a summary table which you can use for quick reference.

In this book, we use the term *literal* to mean either a variable or a Boolean value (e.g., 1 or 0). Elsewhere (for example, in formal logic in artificial intelligence), the term is used for both these (positive literals) as well as for their negation (negative literals). In the following laws, the letters "A", "B", etc., stand for literals or entire subexpressions.

Commutative Law.

`AND` version: $AB = BA$
`OR` version: $A + B = B + A$

The commutative law allows us to change the position of the operands of particular operators. You already know the the commutative law for addition $(3 + 2 = 2 + 3)$. There are also forms of the commutative law governing `AND` and `OR`, as shown above.

Note that we show only one way of writing each of the laws, using juxtaposition of terms for `AND` and $+$ for `OR`. We could have also specified the `AND` form of the Commutative Law, for example, as:

$$A \wedge B = B \wedge A$$

Note that just because many operators obey a commutative law (i.e., they *commute*), do not think that they all do. Subtraction, for

example, is not commutative: $3 - 2 \neq 2 - 3$. The Boolean operators we are concerned with here, though, all commute.

The commutative law is used in proofs to switch the order of operands to match what we are trying to prove or to move operands together that other laws can operate on. For example, suppose that we are trying to prove that:

$$CB + D \equiv D + CB$$

Given the rules of precedence, this is the same as:

$$(CB) + D \equiv D + (CB)$$

We would like to use the Commutative Law to rearrange the left-hand side to look like the right-hand side, but we can't, at least not directly: the `OR` version of the law has to do with variables, not compound things such as (CB).

This is a good time to introduce a trick that will come in handy whenever you are proving things. By letting a new variable, say X, equal the subexpression (CB), we get:

$$X + D \equiv D + X$$

which, by the Commutative Law, is correct. Thus it is also true for the case when X is replaced by the subexpression.

As an example of this and of using the Commutative Law, consider the following. Suppose we are trying to prove that:

$$ABC \equiv CAB$$

We can do this in the following steps:

ABC	Given.
$(AB)C$	Precedence, definition ().
XC	Let $X = (AB)$.
CX	Commutative Law.
$C(AB)$	Substitution for X
CAB	Precedence, definition of ().
	$\therefore ABC \equiv CAB$

This is an actual proof. We start with what we are given, and we end the proof when we have derived the thing to be proved. At each step, we use one law, and we explicitly state what that law is.

Associative Law.

`AND` version: $ABC = (AB)C = A(BC)$
`OR` version: $A + B + C = (A + B) + C = A + (B + C)$

When three (or more) Boolean values are AND'ed or OR'ed together, they are normally evaluated from left to right. That is, the first two associate with each other first. The associative law allows us to change the order in which they are evaluated, that is, to associate different ones together first.

Note that the first transformation in each is not an application of the associative law, but rather is just making explicit with parentheses the existing association between A and B due to precedence rules.

The Associative Law is used in proofs to make an expression look more like what we are trying to prove as well as to group subexpressions that other laws can then operate on. Remember, this law only applies when the operators are of the same type and when the two operations appear together in the expression.

Null Law.

AND form: $0A = 0$
OR form: $1 + A = 1$

Identity Law.

AND form: $1A = A$
OR form: $0 + A = A$

Sometimes we would like to get rid of part of an expression in order to get it to look more like what we're trying to prove. The Null and Identity Laws allow us to simplify an expression with variables by giving instances where the value of that expression can be predicted.

By examining the truth tables for AND and OR, we can see that for both there is a value that dominates the expression whenever it is present. For AND, the result is 0 whenever there is a 0 as an operand. For OR, the expression is 1 whenever there is a 1 as an operand. The other value—the value of the other variable—does not dominate the expression in this way. In fact, its presence does nothing to affect the value of the overall expression. So, if we AND a 1 with some other value, the value of the expression will be that other value—a 1 if the other value is a 1 and a 0 if the other value is a 0. Similarly, if we OR a 0 with another value, that other value will be the value of the whole expression. These ideas are axiomatized in the Identity and Null Laws.

Absorption Law.

AND form: $A(A + B) = A$
OR form: $A + AB = A$

In some situations, a literal dominates the expression because, regardless of its value, the whole expression will have that value. The Absorption Law capitalizes on this to allow us to turn much larger expressions into a single literal in that expression.

For example, in the AND form of the law shown above, if A has a value of 1, then the OR'ed subexpression will have a value of 1, regardless of the value of B:

$$1(1 + B) = 1 \wedge 1 = 1$$

If $A = 0$, then we know from the Null Law that the expression will equal 0—since anything AND'ed with 0 is 0—which is the value of A:

$$0(0 + B) = 0B = 0$$

.
Thus the expression depends only on the value of A.

We can make a similar argument for the OR form.

Idempotent Law.

AND form: $AA = A$
OR form: $A + A = A$

When a value is combined with itself and yields the value itself, the value or operation is said to be *idempotent*. The Idempotent Law allows us to create (or remove) copies of literals.

If we apply the AND or OR operator to two operands that are the same value, the result will be that value. That is, both AND and OR are idempotent. The Idempotent Law allows us to remove duplicate copies of literals, which can be handy in proofs.

So WE HAVE SEEN how to group or rearrange variables when the operations linking them are all the same. What about when they are different? Are there ways we can transform an expression that has different kinds of operators in it, so that we can get it closer to what we are trying to prove?

One law that can help here is the Distributive Law, which allows us to combine operations in a different way. It's called the Distribu-

tive Law because one operator is *distributed* across another to do the transformation.

Distributive Law.

AND form: $A(B + C) = AB + AC$
OR form: $A + BC = (A + B)(A + C)$

The AND form should look very familiar: it is identical to the Distributive Law for arithmetic. The other form of the law is not so familiar. However, if we rewrite it using $\wedge$ and $\vee$, it doesn't look quite so bizarre:

$$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$$

This works for Boolean algebra, whereas it doesn't for everyday arithmetic. Notice that once the operand is distributed, the operations involving that operand are performed first. This is shown with parentheses in the AND form and implied by operator precedence in the OR form.

As an example of the use of the Distributive Law, consider proving the AND form of the Absorption Law: $A(A + B) = A$.

$A(A + B)$	Given.
$AA + AB$	Distributive Law.
$A + AB$	Idempotent Law.
$1A + AB$	Identity Law.
$A1 + AB$	Commutative Law.
$A(1 + B)$	Distributive law.
$A(1)$	Identity Law.
$1A$	Commutative Law.
A	Identity Law.
	$\therefore A(A + B) = A$

Note that although the first occurrence of $1A$ in the proof may have led you to wonder where it came from, it is a straightforward application of the Identity Law—laws can be used in either direction. Here, we are using the law to introduce a 1 into the expression.
As another example, let's see if we can prove the OR form of the Distributive Law using the AND form. We'll start with the right-hand side of the OR form:

$(A + B)(A + C)$	Given.
$X(A + C)$	Substitution of X for $(A + C)$.
$XA + XC$	Distributive Law.
$(A + B)A + (A + B)C$	Substitution.

$AA + AB + (A + B)C$	Distributive Law.
$AA + AB + AC + BC$	Distributive Law.
$A + AB + AC + BC$	Idempotent Law.
$1A + AB + AC + BC$	Identity Law.
$A1 + AB + AC + BC$	Commutative Law.
$A(1 + B) + AC + BC$	Distributive Law.
$A1 + AC + BC$	Null Law.
$A(1 + C) + BC$	Distributive Law.
$A1 + BC$	Null Law.
$A + BC$	Identity Law.
	$\therefore (A + B)(A + C) = A + BC$

Double Negation Law.

$$\overline{\overline{A}} = A$$

This law will likely seem obvious. However, double negation allows us to add or remove negation by asserting that negating an operation twice is the same as not negating it at all. Since there are only two values in Boolean logic, the negation operator works as a toggle between those two values. If you toggle the value once, you change it from 0 to 1 or from 1 to 0. If you toggle it again, you change the value back to the original value.

Inverse Law.

`AND` form: $A\overline{A} = 0$
`OR` form: $A + \overline{A} = 1$

The Inverse Law allows us to simplify an expression when the operands of an operator are a value and its inverse. For both operators, we know what the value of the operation is if the operands are a value and its inverse. We can remember these values by thinking about the null and identity laws. For `AND`, either the A or its inverse will be 0. If A is 0, then the Null Law tells us that the value of the operation will be 0. If A is 1, then the inverse of A is 0. We can use the Commutative Law to change the order of the operands making the inverse of A the first operand. Now, the first operand is 0 and the Null Law, once again, can be applied. We can make a similar argument for `OR`.

De Morgan's Law.

NAND/OR version: $\overline{AB} = \overline{A} + \overline{B}$
NOR/AND version: $\overline{A + B} = \overline{A}\,\overline{B}$

De Morgan's Law allows us to convert between AND and OR operations. On the left hand sides of the equations above, the result of the operation will be negated. On the right hand sides, each operand is negated and the operation has been changed, from AND to OR or vice versa. If we want to apply De Morgan's Law but the operation has not been negated, we can use the law of double negation to negate the operation (twice) and then apply De Morgan's Law, for example, applied to AB:

AB	Given.
$\overline{\overline{AB}}$	Double negation.
$\overline{\overline{A} + \overline{B}}$	De Morgan's Law.

Note that although it is tempting, it doesn't gain us anything to apply De Morgan's Law again; if we do, we get:

$\overline{\overline{A} + \overline{B}}$	Given.
$\overline{\overline{A}}\,\overline{\overline{B}}$	De Morgan's Law.
$A\overline{\overline{B}}$	Double negation.
AB	Double negation.

which gets us back to where we started.

As an example of using De Morgan's laws, prove that $\overline{\overline{A}\,\overline{B} + C} \equiv (A + B)\overline{C}$.

$\overline{\overline{A}\,\overline{B} + C}$	Given.
$\overline{X + C}$	Substitution of X for $\overline{A}\,\overline{B}$
$\overline{X}\,\overline{C}$	De Morgan's Law.
$\overline{\overline{A}}\,\overline{\overline{B}}\,\overline{C}$	Subtitution.
$(\overline{\overline{A}} + \overline{\overline{B}})\overline{C}$	De Morgan's Law.
$(A + B)\overline{C}$	Double Negation Law, twice.
	$\therefore \overline{\overline{A}\,\overline{B} + C} = (A + B)\overline{C}$

All of the laws we have seen so far can be summarized in Table 5.2.

Proof

The overall goal is to be able to take the laws of Boolean algebra and apply them to transform one Boolean expression into another. This

Commutative Law:	$AB = BA$ $A + B = B + A$
Associative Law:	$(AB)C = A(BC)$ $(A + B) + C = A + (B + C)$
Absorption Law:	$A(A + B) = A$ $A + AB = A$
Null Law:	$0A = 0$ $1 + A = 1$
Identity Law:	$1A = A$ $0 + A = A$
Idempotent Law:	$AA = A$ $A + A = A$
Distributive Law:	$A(B + C) = AB + AC$ $A + (BC) = (A + B)(A + C)$
Double Negation:	$\overline{\overline{A}} = A$
Inverse law:	$A\overline{A} = 0$ $A + \overline{A} = 1$
De Morgan's Law:	$\overline{AB} = \overline{A} + \overline{B}$ $\overline{A + B} = \overline{A}\,\overline{B}$

Table 5.2: Summary of Boolean algebra laws.

may be to transform a given expression into one that is minimal or to prove that one expression is equivalent to another. If we are considering applying a law to transform expression 1, let's say, then if we are very lucky, expression 1 is in a form that exactly matches the pattern on one side or the other of the $=$ in the law. For example, if our expression is $W(X + Y)$, then this matches the general form of the Distributive Law, with W being matched to A, X to B, and Y to C in the statement of the law given above.

This will not always be the case, however. Suppose we are given $(X + Y)W$. Intuitively, we can see that we can still apply the Distributive Law, but technically we can't unless the expression matches one of the patterns directly. In this case, what we really need to do is to move W to the left side of the expression. But can we? We have no law with a pattern that looks exactly like that.

The trick here, as you probably have already figured out, is to treat

the pieces of an expression—the sub-expressions—as chunks. That is, if we let a new variable, say Q, be equal to $(X + Y)$, then we have QW. This matches the AND form of the Commutative Law, and so we can apply the law to give WQ. Now, substituting the value of Q, we get $W(X + Y)$, which now matches one of the patterns for the Distributive Law. The renaming of sub-expressions (e.g., $(X + Y) = Q$) need not be written down in the proof, although it can be—and probably should be, at least while one is learning to write proofs.

Once we understand how we will rename subexpressions as variables that can match the literals in the laws as given, we can apply those laws in proofs. To apply a law, the renamed expression needs to match exactly to one side of the equation given in the law. Since equality is symmetric (if $A = B$ then $B = A$), it does not matter whether the left or right side of the equation matches the current expression. In either case, we match the current expression to one side of the law's equation and replace it with the other side of the equation. The renaming scheme that allows us to match one side of the equation also needs to be applied to the other. This method of replacing one expression with an equivalent one is called substitution. To help the reader of the proof understand what has been done, we write the name of the law that was applied to the right of the new expression.

For example, consider proving that

$$(XZ + Y)W = WX + WYZ$$

Let's start with the left-hand side and see if we can derive the right hand side, using only valid laws of Boolean algebra. We first write down our starting point:

$$(XZ + Y)W \quad \text{Given}$$

Although we will write the proof following the changes in the left-hand side, we can think about the proof in any way that is helpful to us. This equation reminds us of the Distributive Law—it looks like it, except the operands at the top level are in the reverse order, and there is an expression (XZ) where we find only a variable in the Distributive Law. So we will have to apply other laws in order to get the expression into a form that matches the Distributive Law.

Let's now see if we can get this into a form where the single variable that is AND'ed with the parenthetical expression is on the left, rather than on the right. We need to use the Commutative Law, but the expression as it stands does not match it. So we will rename things in it until it matches:

$$AW \quad \text{Let } A = (XZ + Y)$$

Here, we create a new variable, A, and let it name the parenthetical expression.

Now we have an expression on which we can use the Commutative Law:

$$WA \quad \text{Commutative Law}$$

Now we rename it back to the way it was:

$$W(XZ + Y) \quad \text{Substitution for } A$$

Our next task is to get this expression into a form that we can use Distributive Law on. It's almost there, except for the expression XZ where we expect a single variable. So we rename things again:

$$W(B + Y) \quad \text{Substitution } B = XZ$$

Now we can apply the Distributive Law (`AND` form):

$$WB + WY \quad \text{Distributive Law}$$

Undoing the substitution, we have:

$$WXZ + WY \quad \text{Substitution}$$

We're almost there, but our subexpressions are again out of order, and we cannot use the Commutative Law as it stands. So we rename again, two parts this time:

$$C + D \quad \text{Let } C = WXZ \text{ and } D = WY$$

Now we can use the Commutative Law to switch the order of the =OR={}ed subexpressions (the disjuncts):

$$D + C \quad \text{Commutative Law}$$

Once we undo the prior substitution, we are done:

$$WY + WXZ \quad \text{Substitution}$$

We have successfully proven what we set out to prove. Table 5.3 shows the proof in its entirety.

Table 5.3: A proof that $(XZ + Y)W = WX + WYZ$.

$(XZ + Y)W$	Given
AW	Let $A = (XZ + Y)$
WA	Commutative Law
$W(XZ + Y)$	Substitution for A
$W(B + Y)$	Substitution $B = XZ$
$WB + WY$	Distributive Law
$WXZ + WY$	Substitution
$C + D$	$C = WXZ, D = WY$
$D + C$	Commutative Law
$WY + WXZ$	Substitution
	$\therefore (XZ + Y)W = WY + WXZ$

Proofs are important because they allow computer scientists, mathematicians, logicians and others to convince people who might use their ideas that those ideas are sound. It is important to learn to write proofs even if we expect to convince others of our big ideas in some other way. Learning to write proofs helps us to develop our skills in formal thinking. Writing a good proof forces us to think rigorously and to commit to each step in writing—we are not allowed to let vague thoughts remain in our argument with the hope that someone else will ignore the details or fill them in for us. Proofs also

help us to develop intuition about the system that we are writing proofs about. This intuition then helps us to work within the system in all sorts of ways, including coming up with the great ideas which we need to prove to others.

Creating a proof begins with some intuition about how the things that you want to prove are related. In our small proof above, our intuition was that we were reminded of the Distributive Law. We work from our intuition to mold one of the sides of the equation to fit the laws that capture that intuition. For example, in the proof above, we had to change the order of the operands to get to the required result.

Sounds easy, right? So, why do so many people complain about writing proofs or just never seem to get it? When proving advanced concepts, new ideas in mathematics, for example, the person may simply not have enough intuition about the concept to create the proof. (There is also the possibility that the new idea is flawed, so it cannot be proved.) Most students, however, have the ability to create proofs for what they are asked to prove. Most often, they complain because creating proofs is frustrating work that may take several attempts, and they fail because they give up too soon. If you realize this before you start to write a proof, and give yourself plenty of time, you are more likely to see writing a proof as an interesting challenge instead of a nightmare homework assignment. You might even have fun, and you are sure to be proud of your work when you have completed it.

Sidebar 5: XOR and equality

Now is a good time for an aside about the relationship between XOR and equality. XOR is true when one or the other of its inputs is true, but not both. The equality function is true when both of its inputs are true or both are false. A little thought will convince you that each of these operators is the negation of the other. But can we prove it?

We *could* prove this via a truth table., but let's instead try proving it via algebraic substitution. XOR, for two inputs A and B, is defined by the expression $A\overline{B} + \overline{A}B$. Equality is defined as $AB + \overline{A}\,\overline{B}$. Can we negate one of these, then prove that it is equivalent to the other?

Let's try negating equality and proving that that is equal to XOR. (We will skip any very obvious steps in this proof.)

$\overline{AB + \overline{A}\,\overline{B}}$	Given
$\overline{AB}\,\overline{\overline{A}\,\overline{B}}$	De Morgan's Law
$(\overline{A} + \overline{B})(A + B)$	De Morgan's Law, twice
$(\overline{A} + \overline{B})A + (\overline{A} + \overline{B})B$	Distributive Law
$\overline{A}A + \overline{B}A + \overline{A}B + \overline{B}B$	Distributive Law
$\overline{B}A + \overline{A}B$	Inverse Law, twice

$A\overline{B} + \overline{A}B$	Commutative Law
	$\therefore \overline{AB + \overline{A}\,\overline{B}} = A\overline{B} + \overline{A}B$

So where do you find the intuition that allows you to create a proof? Sometimes, the answer jumps right out at you. You may be reminded of a law, as above. Or what you are trying to prove may have a meaning that you "know" is right. You will see an example of that below. These intuitions provide the core of your proof.

As discussed above, if your intuition reminds you of a law, you must manipulate the expressions so that law can be applied. If your intuition is about the meaning of the equation and why these expressions should be equal, you must formalize your idea through applicable laws.

For example, suppose you are asked to prove $ABCD = CBDA$. Your intuition may tell you that this should be true: we can change the order of two operands that are `AND`'ed together, and here we seem to simply be changing the order over and over again. We use this intuition to recognize that we must apply the Commutative Law several times to change the order of the operands.

Our proof is:

$ABCD$	Given.
$BACD$	Commutative Law.
$BCAD$	Commutative Law.
$CBAD$	Commutative Law.
$CBDA$	Commutative Law.
	$\therefore ABCD = CBDA$

Sometimes, unfortunately, you have no intuition about how to begin the proof. In that case, you can try manipulating one or the other side of the equation to see if you can see relationships between the newly-generated expressions and the expressions that you have been given. Be sure to keep track of how the new expressions were formed, since you will need to include that information in your proof.

If you still cannot find a way to start, don't worry. Take a break. Just getting away from the work for a while can help you find a way to start the proof when you come back. If you go for a walk, go to the gym, fold laundry or do some other task that allows you to think without *forcing* you to think about the proof, you may find that your mind wanders back to the proof and to some ideas for how to begin.

A proof is valid if it proves what needs to be proven in a convincing way. However, just as we continue to edit a paper after the first draft, many proofs can be improved. Proofs are improved by making

them more elegant. This means that the proof is simple and uses an intuitively appealing argument.

Once you have completed a proof, you should make sure that it is valid and convincing. For the proofs that are discussed in this section, you can do this by checking all of the steps and making sure that they give the proper rule as justification and use that rule correctly. Next, you can try to find other ways of forming a proof. The work that you have done to create the first proof may give you an insight that helps you to see a new approach to the proof. You can also study the existing proof with an eye toward eliminating unnecessary steps or proving sections of the proof more elegantly.

It can be very frustrating to work on a proof, or on any kind of hard problem. Here are some suggestions for dealing with that frustration. Some are specific to proofs, but most can be applied to any problem which requires a lot of thinking before seeing how to answer the questions:

Remember that it is not unusual to make several attempts before you find the right basic argument for your proof.

- Remember that you often have to manipulate an expression to get it into the form that allows you to apply a particular rule.
- Write things down on scratch paper and use what you need to create the proof.
- There are also ways to deal with frustration that can be applied to problem solving and programming in general:
  - Take a break and come back.
  - Work with others, if allowed by your instructor, to brainstorm and refine solutions.
  - Break the problem into small parts. Work on parts separately and then put them back together.
  - Explain the problem to someone who does not have expertise in the area to help you clarify your thinking.

Summary

This chapter focused on Boolean algebra, an algebra for two-valued logic. You have learned some ways of proving equivalence between two Boolean algebra expressions, including truth table-based proofs and proof by algebraic substitution. While the former is arguably easier, we saw that there are two cases where it either should not or cannot be used: when the number of variables is large and when we do not know what the expression is to which we are trying to prove equivalence (e.g., in minimizing an expression/circuit).

The chapter also covers some of the laws of Boolean algebra. We saw how to apply these laws to do algebraic substitution.

In the next chapter, we will see how to create logic circuits from Boolean algebra specifications.

Review

Further reading

Exercises

1. Prove both the AND and OR forms for the following laws by constructing truth tables:
   (a) Identity Law
   (b) Idempotent Law
   (c) Distributive Law
   (d) DeMorgan's Law
2. We can prove that $A \oplus B \equiv \overline{A}B \ + \ A\overline{B} \equiv \overline{\overline{(\overline{A}B)} \bullet \overline{(A\overline{B})}}$. Now prove that $A \oplus B$ is also equal to $\overline{\overline{A}B} \ + \ \overline{A\overline{B}}$.
   For the proof, use algebraic manipulation to show that the expression given is equivalent to one of the expressions shown in class to be equivalent to $A \oplus B$.
3. Using algebraic manipulation, prove the OR form of the absorption law, $A \ + \ AB \ = \ A$.
4. Using algebraic manipulation, prove that $A\overline{B}C \ + \ ABC \ = \ AC$.
5. We are trying to design a circuit for the *majority function* which has an output of 1 when most of its inputs are 1. Our circuit will have three inputs. One expression for the function is $AB\overline{C} \ + A\overline{B}C \ + \ \overline{A}BC \ + \ ABC$. What is the expression with the minimum number of AND, OR, and NOT gates and number of inputs inputs that you can find? Prove that your expression is equivalent to the expression that you have been given.
6. We are trying to design a circuit that has five inputs and has an output of 1 whenever an odd number of inputs are 1. Write an algebraic expression that captures the function. You do not need to minimize the expression in any way.
7. You might wonder, for n inputs, how many possible different Boolean functions there are. For example, for one input, there are only four, with these truth tables:

Input	F_1	F_2	F_3	F_4
0	0	0	1	1
1	0	1	0	1

   (a) Write the Boolean algebra functions for F_1–F_4.
   (b) (Advanced) Determine a general formula for how many functions are possible for n input variables. How many functions are possible with 2 inputs? With 3? With 4?

HINT: You can see how we got 4 functions for a single input. There are 2 rows in the table for a single input. That means that, reading down the function output column, there are two binary digits. With two binary digits, you can represent $2^2 = 4$ different combinations, and thus, different functions.

1. Prove both the AND and OR forms for the following laws by constructing truth tables:
   (a) Identity law
   (b) Idempotent law
   (c) Distributive law
   (d) DeMorgan's law

2. We can prove that $A \oplus B \equiv \overline{A}B \ + \ A\overline{B} \equiv \overline{\overline{(\overline{A}B)} \bullet \overline{(A\overline{B})}}$. Now prove that $A \oplus B$ is also equal to $\overline{\overline{A}B} \ + \ \overline{A\overline{B}}$. For the proof, use algebraic manipulation to show that the expression given is equivalent to one of the expressions shown in class to be equivalent to $A \oplus B$.

3. Using algebraic manipulation, prove the OR form of the absorption law, $A \ + \ AB \ = \ A$.

4. Using algebraic manipulation, prove that $A\overline{B}C \ + \ ABC \ = \ AC$

5. We are trying to design a circuit for the *majority function* which has an output of 1 when most of its inputs are 1. Our circuit will have three inputs. One expression for the function is $AB\overline{C} \ + A\overline{B}C \ + \overline{A}BC \ + \ ABC$. What is the expression with the minimum number of `AND`, `OR`, and `NOT` gates and number of inputs inputs that you can find? Prove that your expression is equivalent to the expression that you have been given.

6. We are trying to design a circuit that has five inputs and has an output of 1 whenever an odd number of inputs are 1. Write an algebraic expression that captures the function. You do not need to minimize the expression in any way.

7. You might wonder, for n inputs, how many possible different Boolean functions there are. For example, for one input, there are only four, with these truth tables:

Input	F_1	F_2	F_3	F_4
0	0	0	1	1
1	0	1	0	1

   (a) Write the Boolean algebra functions for F_1–F_4.
   (b) (Advanced) Determine a general formula for how many functions are possible for n input variables. How many functions are possible with 2 inputs? With 3? With 4?
   HINT: You can see how we got 4 functions for a single input. There are 2 rows in the table for a single input. That means that, reading down the function output column, there are two binary digits. With two binary digits, you can represent $2^2 = 4$ different combinations, and thus, different functions.