Digital Logic

Foundations of Computer Science

E.H. Turner and R.M. Turner

2017-05-24.1

This part of the book is about how, at a very basic level, a computer does what it does. In Chapter **??**, we introduced the idea of digital representation of data (and programs) and, hence, digital hardware. Before talking in depth about programs such as operating systems, before talking about programming languages, and even before talking about the major components of computers and how they work together, we must first understand at the most basic level what this digital hardware is and how it operates. Toward this end, we look in this part of the book at something called *digital logic*, which is the underpinning of computers.

An understanding of digital logic is fundamental to an understanding of computers, and so this part of the book is different from most other sections in that there are not one, but three, introductory chapters, followed by three in-depth chapters. The current chapter introduces digital logic, including the concepts of logic gates, circuits, and computer chips. The following chapters present Boolean algebra, digital logic circuits and how they can be specified by Boolean functions, and circuit minimization—that is, how a given function can be realized with the minimum number of gates—and covers a fundamental technique for doing so, Karnaugh maps. After that, the digital logic part of the book concludes with two chapters discussing the two major kinds of logic circuits, combinational and sequential, by focusing on useful example circuits, adders and registers, respectively.

What is digital logic?

The term *digital logic* is commonly used in two ways. First, it is used to refer to the basic physical components of the computer that allow it to process and remember information. Second, it is used to mean the set of laws and operations that govern the manipulation of binary information. We will look first at the latter, then see how we implement digital logic in hardware.

Boolean algebra

Major components of the computer, and indeed the computer as a whole, can be thought of as computing functions on a set of inputs.

We need to take a look at what these functions are and how they are computed. To do this, we need a way of describing them, which for computers is *Boolean algebra*, a type of mathematical language.

Regular algebra uses numbers and variables on which operators, well, operate. Instead of numbers, Boolean algebra deals with operators that work on true/false values, also called *Boolean values*. Evaluating a Boolean algebra expression involves determining the truth or falsehood of an expression composed of true/false values and Boolean operators.

For example, consider the statement: "It is raining and the temperature is not below freezing". This can be represented in Boolean algebra terms as the *conjunction* of two statements, "it is raining" and "the temperature is not below freezing". The latter statement, in turn, can be considered equivalent to the negation of the statement "the temperature is below freezing". If we call the first statement A and the second B, then we can represent this whole sentence as the Boolean algebra statement: A `AND NOT` B.

Digital computers operate on two values as well, 0 and 1,[1] so we can see that there is a correspondence between Boolean values and the computer's binary values, with 0 corresponding to "false" and 1 to "true". Thus, computers and logic circuits in general naturally lend themselves to being described by Boolean algebra.

[1] Actually, the inputs and outputs of computers, as well as internal states of components, are voltage levels, for example, in the range of 0–3.3 V (volts) or 0–5 V. Voltage ranges are divided such that voltages below some value means "0", while voltages above some value mean "1".

We will have much more to say about Boolean algebra in the next few chapters, where we discuss how it can be used to describe the computations performed by digital circuits on their inputs to yield outputs. In this chapter, we are concerned with the *Boolean operators* themselves and some simple expressions composed using them. These operators describe how values in the computer are operated on to carry out the computer's functions.

Truth tables

Although we can describe Boolean operators in English, we really need a more concise, exact way to describe them that is not open to different interpretations. For this, we use *truth tables*.

A	B	F
0	0	1
0	1	0
1	0	0
1	1	1

A truth table contains a column for each of an operator's *operand* and one for the result of performing the operation. The header row contains the names we choose to give the operators and the result (the actual names do not matter). Each row lists a particular combination of values for the operands as well as the result of carrying out the operand on those values. We include a single row for each unique combination of values for the operands. Taken together, then, the rows of the truth table describe the results for every possible combination of operands.

For example, the truth table in the margin tells us the result (labeled F) of applying the operator to its operands. We also call operands *inputs* or *variables*, and we will often refer to the result as the *output* of the operator when given those sets of inputs. This truth table tells us, for example, that for the inputs $A = 1$, $B = 0$, the output is 0, i.e., $F = 0$. Note that F is a *Boolean function*.

As you may have already guessed, truth tables can be used for more than just single operators. Combinations of operators and their operands—i.e., *Boolean expressions*—can also be represented this way, since each will have a number of variables, each with two possible values. The truth tables will be in general much bigger, as we will see later, but the principle is exactly the same: for every possible combination of input values, list the output (i.e., the value of the Boolean expression).

We will see later that there is a well-defined relation between the number of input columns and the number of rows in a truth table. We will also see that there is a standard way to order the rows.

Boolean operators

Let's first look at three common Boolean operators, AND, OR, and NOT. These are important, since together they form a *complete operator set*: that is, with these three, we can create expressions for any Boolean function—and hence, any digital logic circuit.

NOT

The simplest operator is NOT, or *negation*. The truth table for NOT is shown to the side. NOT has only one operand, which we call here A. We can think of A as standing for a statement, such as "It is raining." The statement is either true (1) or false (0). The output, now labled with the *expression* NOT A rather than a symbol, tells us the truth value for both possible values of A. In the first row, where A is false, then "NOT A" is true: if "It is raining" is false, then "It is not raining" is true.

A	NOT A
0	1
1	0

For the rest of the operators, we will briefly discuss their meaning and then give their truth tables. It may be helpful to you to create your own examples like the one above.

`AND`

The `AND` operator follows our intuition about what "and" means in English. The word "and" lets us join two things together (i.e., it is a *conjunction*); the conjoined statement is only true if both things are true. For example, the statement "It is raining and Obama is president" is only true if it is both raining and if Obama is indeed president.

This is also true of the conjunction operator `AND`, as the truth table in the margin shows. Note that the only row in which the conjunction is true is the one in which both inputs are true.

A	B	A `AND` B
0	0	0
0	1	0
1	0	0
1	1	1

`AND` is an example of a *binary operator*, whereas `NOT` is a *unary operator*. This can be confusing, since we are also talking about binary numbers. Here, though, "binary" just means that the operator takes two operands.

`OR`

The `OR` operator is another one that at first glance seems to follow our everyday use of the word "or". However, it is somewhat different than we may expect. In English, if we say "It is raining or it is snowing", we usually mean either it is raining or it is snowing, but not both. If "Dan or Phil is driving", for example, we would be very surprised if *both* of them had the wheel. Our normal use of the word "or" actually corresponds to the logical operation `XOR`, or *exclusive or*, which we discuss below.

A	B	A `OR` B
0	0	0
0	1	1
1	0	1
1	1	1

Logical (or Boolean) `OR` is the *inclusive or*: one or the other, or both, of the things `OR`'d together are true, as the truth table shows.

Boolean expressions

Boolean operators, like arithmetic operators, are not very useful by themselves, but rather when they are combined into expressions. As in ordinary algebra, an expression is a statement written as a combination of operators and operands that has an overall value. In algebra, the operators are such things as $+$, $-$, $\times$, and $\div$, and the operands are numbers or variables. Parts of the expression—subexpressions—can be grouped using parentheses.

Boolean expressions use Boolean operators, constants (true or false), variables, and parentheses. The overall value of the expression is one of the two Boolean values, true or false (i.e., 1 or 0).

As in algebra, instead of writing out the name of a Boolean operator, we usually use symbols. Unfortunately, Boolean algebra differs from algebra here: there are several sets of symbols in use for the Boolean operators. Table 1 shows some common symbols used.

Operator	Examples			
NOT	$\sim A$	$\neg A$	$\overline{A}$	
AND	$A \times B$	$A \wedge B$	$A \cap B$	AB
OR	$A + B$	$A \vee B$	$A \cup B$	

Table 1: Common symbols for Boolean operators

It can be easy to confuse the symbols we typically use in this book for AND ($\wedge$) and OR ($\vee$), since they are so similar. One way to remember which is which is that the $\wedge$ symbol looks a little bit like an "A", the first letter of "and".

Note that AND is sometimes represented by the intersection symbol ($\cap$) and sometimes in the same way we represent multiplication; and OR is sometimes represented by the union symbol ($\cup$) and sometimes like we represent addition. There are good reasons for this, since there is similarity between AND, intersection, and multiplication, and between OR, union, and addition. We won't belabor this point here, but you may see some similarities with familiar operators when we talk more about Boolean algebra in Chapter **??**.

Now, armed with this knowledge of how to write Boolean operators, you should have no trouble writing or understanding the meaning of Boolean expressions. For example, suppose we want to determine if we should take an umbrella when we leave home in the morning. We should take an umbrella if it is raining as we are leaving. We should also take an umbrella if our not-very-reliable weatherman said it will rain and it looks cloudy. We can create a function that captures the reasoning we use to make a decision.

For this example, let's use R to mean that it is raining, W to mean that the weatherman predicted rain, and C to mean that it is cloudy. We will let U be the value of the expression. If $U = 1$ (i.e., "true"), then we will take an umbrella, otherwise we will not.

To write the expression U is equal to, we must consider the statement of the problem. We will take an umbrella ($U = 1$) if either or both of the following conditions are true: (1) it is raining ($R = 1$); or (2) both the weatherman predicted rain ($W = 1$) and it is cloudy ($C = 1$). This can be written as:

$$R \texttt{ OR } (W \texttt{ AND } C)$$

or, in symbols:

$$R \vee (W \wedge C)$$

If we set U equal to this expression, we have a *Boolean equation*:

$$U = R \vee (W \wedge C)$$

As you can see, R, W, and C are input (independent) variables, and U is the output (dependent) variable. We could write this with U being a *function* of R, W, and C:

$$U(R, W, C) = R \vee (W \wedge C)$$

In this book, we will usually dispense with the function notation.

On any particular day, we assign appropriate values to the inputs to yield the value of the output U. We can show all possible combinations of inputs and the corresponding output using the truth table shown.

R	W	C	U
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Suppose on one particular day the weatherman predicted rain: $W = 1$. We look out the window and see that it is not raining ($R = 0$) or cloudy ($C = 0$). Using these values for the variables, we can look up the corresponding row in the truth table, in this case, the third row. We can then immediately see from the output column that $U = 0$, i.e., we do not need an umbrella.

Though easy to use, it is not particularly clear from this truth table how we arrived at the values for U. To fix this, we can expand our truth table to include intermediate columns for subexpressions, and we can also label the output with the entire expression:

R	W	C	$W \wedge C$	$U = R \vee (W \wedge C)$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Here, the intermediate column is easy to compute, which helps us compute U, and it also helps the reader understand and verify our result.

Expressions can be much more complex than this, of course. For example, later you will be able to prove that:

$$R \vee (W \wedge C) \equiv \overline{\overline{R} \wedge \overline{(W \wedge C)}}$$

By combining operators, we can come up with arbitrarily-complex expressions. In fact, we can use Boolean expressions to represent anything that can be computed.

We should point out two advantages Boolean expressions have over English (or any other natural language) for representing logical assertions (and, hence, digital logic). First, Boolean expressions are much more concise than their natural language counterparts. Second, Boolean expressions are not ambiguous; the same cannot be said for natural languages. For example, if we say "it is not raining or the weatherman predicted rain and it is cloudy", we are not completely sure whether this means $(\overline{R} \vee W) \wedge C$ or $\overline{R} \vee (W \wedge C)$; these are different as can be seen in Table 2.

Table 2: See text.

R	W	C	$(\overline{R} \vee W) \wedge C$	$\overline{R} \vee (W \wedge C)$
0	0	0	0	1
0	0	1	1	1
0	1	0	0	1
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Precedence

Just as in ordinary algebra, there are *precedence rules* for Boolean algebra, and they are very similar. Subexpressions inside parentheses are evaluated first, then `NOT`, then `AND` and `NAND`, and finally `OR`, `NOR`, and `XOR`.

Logic gates

Boolean logic would not be very useful for us if there were not a way to *implement* Boolean expressions in digital logic. Fortunately, there are electronic devices, called *logic gates*, that correspond to the Boolean algebra operators we have just discussed: `NOT` gates, `AND` gates, and `OR` gates. Input *lines* (wires) entering the gate represent the corresponding operator's operands, and an output line leaving the gate represents the result. The Boolean values 0 and 1 of the variables are represented by applying different voltages to the lines. By wiring the outputs of gates to the inputs of others, we can create a *logic circuit* that can compute more complicated Boolean expression—in fact, we can do this to create the extremely complex circuit that is a complete CPU.

For the most part, we will not worry about how gates are constructed (but see Sidebar 1 if you're interested), their component transistors, resistors, etc., are below the abstraction level in which we are interested. Considering them each time we think about gates would only be a distraction from understanding digital logic. Instead, we will almost always treat gates as "black boxes" (see Figure **??**) with inputs and outputs, and we will consider the values of the variables (values on the lines) as being 0s or 1s, not voltages.

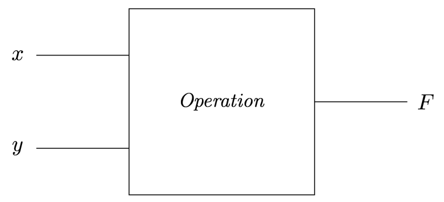


Figure 1: Generic representation of a logic gate.

To differentiate different gates, we will use different shapes for the "box", as we shall see.

Basic gates

We will first look at three basic gates that correspond to the Boolean operators we have seen. Figure 2 shows the symbols we use to represent these gates. Their truth tables are, of course, identical to those of the corresponding operator.

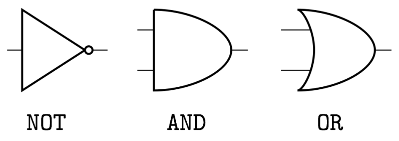


Figure 2: Common symbols for NOT, AND, and OR gates

In the figure, inputs come in from the left and outputs exit on the right. Of course, the gates can be turned other directions, but the inputs and outputs will still be in the same relative position on the gates. The gates shown have at most two inputs and one output. `NOT` is a *unary* gate, while `AND` and `OR` are *binary* gates. Later, we will see that some gates can have more inputs than this, but for now, we'll stick with these.

When we draw circuits containing multiple gates, the gates are connected by lines that stand for the real wires connecting the corresponding electronic devices. When considered at this level of abstraction, each line can have either the value of 1 or 0.

These three gates form a *functionally complete*set of gates that can be the basis for digital logic. That is, the operations `NOT`, `AND`, and `OR` are sufficient to implement any Boolean algebra expression and, hence, any logic circuit.

Other gates

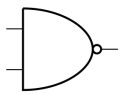
There are gates other than `NOT`, `AND`, and `OR` that are important for computers. The three we will mention here are `NAND`, `NOR`, and `XOR`.

1. `NAND`

   The `OR` gate outputs a 1 if at least one of its inputs is true. Sometimes, we may want to determine if at least one of two inputs variables is false instead. For this, we can use a `NAND` gate.

   `NAND` means "not and", and it indeed is the same as if we were to combine the two Boolean operators (or gates) `NOT` and `AND`. The truth table and symbol for `NAND` are:

A	B	A `NAND` B
0	0	1
0	1	1
1	0	1
1	1	0

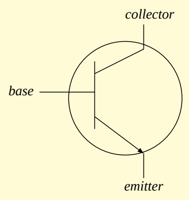


   Note that this is precisely the same truth table as you would get for the expression $\overline{AB}$, as you would expect. Also note that the symbol for `NAND` looks very similar to the symbol for `AND` except for the little circle on the left. As we will see later, a small open circle like this is another way to indicate negation (i.e., `NOT`).

Sidebar 1: What are gates, really?

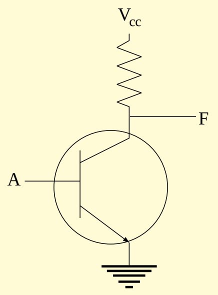
We have so far treated gates as black boxes that compute Boolean functions. But what are they? That is, how are they *implemented* as electronic circuits?

Gates are composed of *transistors*. (Older gates were built from vacuum tubes.) A transistor is a semiconductor device that is able to act as (among other things) a switch. It has two inputs and an output, the collector, base, and emitter, respectively, as shown to the right.

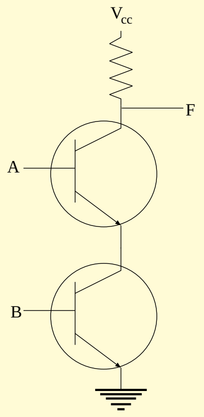


When no voltage is applied to the base, there is no current flow from the collector to the emitter. When a small voltage is applied to the base, current flows from the collector to the emitter.

The simplest logic gate is the NOT gate, which can be created with a single transistor, as shown at the right. When input A has no voltage, representing logical 0 (false), then no current flows through the transistor. Consequently, no current flows through the resistor (the squiggly line above the transistor), either. Although we won't go into it in more detail, this means that there is no voltage drop across the resistor, and so the voltage at output F is the same as V_{cc} (which is called the common collector voltage, which represents a logical 1 [i.e., true]). If we apply voltage to A, representing $A = 1$, then the transistor is switched on, and the voltage at the output F will be 0, the same as ground (represented as the triangular set of lines at the bottom of the figure), which is logical 0. Thus when $A = 1$, $F = 0$ and vice versa, so the circuit represents $\overline{A}$.



A NAND gate can be constructed by putting two transistors in *series*, as shown at right.that is, with the emitter of one hooked to the collector of the next. The two inputs of the NAND gates are then connected to the bases of the transistors.



Here, current only flows to ground, and hence, F is only 0, when both transistors are switched on—that is, when both $A = 1$ and $B = 1$. If either is off (0), then there is no current through that transistor, and hence no current to ground, which means that there is no voltage drop across the resistor, and the output will be 1. This is of course the definition of $\overline{AB}$. One of the exercises asks you to design a NOR gate using transistors.

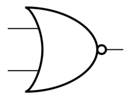
Just as there are several ways to write the AND operator (×, ∧, juxtaposition), there are several ways to write NAND as well in Boolean expressions, for example: A NAND B, $A|B$, $A \uparrow B$, or just $\overline{AB}$.

NAND has the interesting property that it by itself is functionally complete, i.e., any Boolean function can be computed using only NAND operators/gates.

2. NOR

   The NOR gate is to OR as NAND is to AND: i.e., its negation. NOR stands for "not or", and the gate outputs a 1 only if *both* of its inputs are 0. Its truth table and symbol are:

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0



   We can see that $A \texttt{ NOR } B \equiv \overline{A + B}$. Again note the similarity between the NOR and OR symbols and the presence of the circle indicating negation. NOR is also functionally complete by itself, as discussed in Sidebar 2.

   We can write NOR in Boolean expressions several different ways, for example, $A \texttt{ NOR } B$, $A - B$, or $\overline{(A + B)}$.

Sidebar 2: Complete operator sets: NOR

We claim in the text that either NAND or NOR can do the work of NOT, AND, and OR. One way to prove this is to prove that, say, NOR, by itself can be used to compute OR, AND, and NOT. If we can do that, then since those three gates can implement any Boolean function, we will have shown that NOR, can, too.

The easiest to implement is NOT. This can be done by NOR'ing a variable with itself. Thus, we claim that:

$$\overline{A} \equiv A \texttt{ NOR } A$$

This can be proven simply, by using a truth table:

A	A NOR A	$\overline{A}$
0	1	1
1	0	0

Here, we see that the last two columns are identical. This means that the two corresponding functions, A NOR A and $\overline{A}$ are *functionally equivalent*, which proves that we can implement NOT with NOR. We will have more to say about functional equivalence in Chapter **??**.

Implementing the other two gates with NOR is somewhat more complicated. For example, for OR, it seems intuitive that we should be able to compute, say, A NOR B, then negate that to get $A + B$, that is:

$$A + B \equiv \overline{(A \texttt{ NOR } B)}$$

which, using the prior identity to replace the `NOT`, becomes

$$A + B \equiv (A \texttt{ NOR } B) \texttt{ NOR } (A \texttt{ NOR } B)$$

Let's use a truth table to see if our intuition is correct:

A	B	A `NOR` B	$\overline{A \texttt{ NOR } B)}$	$A + B$
0	0	1	0	0
0	1	0	1	1
1	0	0	1	1
1	1	0	1	1

Comparing the last two columns, we see that our approach does indeed work.

Finally, let's see if we can perform `AND` using only `NOR`. In Chapter **??**, we will see how to derive the expression we need. Here, we will simply state that: $AB \equiv \overline{\overline{A} + \overline{B}}$.

Now, $\overline{\overline{A} + \overline{B}}$ is just another way to write $\overline{A}$ `NOR` $\overline{B}$, so $AB = \overline{A}$ `NOR` $\overline{B}$.

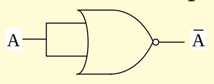
The truth table that proves this is:

A	B	$\overline{A}$	$\overline{B}$	$\overline{A}$ `NOR` $\overline{B}$	AB
0	0	1	1	0	0
0	1	1	0	0	0
1	0	0	1	0	0
1	1	0	0	1	1

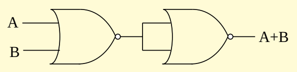
Since we already know how to compute $\overline{A}$ using only `NOR`, we're all set.

Thus we have shown that `NOR` is a sufficient basis for digital logic all by itself. Something similar can be done to show that `NAND` is sufficient, as well.

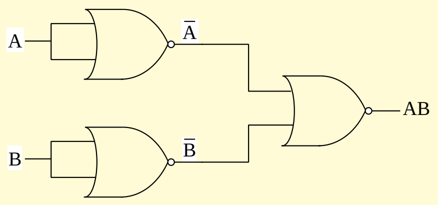
The circuits corresponding to the above expressions are:



`NOT`



`OR`



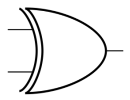
`AND`

3. `XOR`

The final gate we will discuss is the `XOR`, or *exclusive or*, gate. Recall that when we introduced `OR`, we mentioned that it was the "inclusive or": it is true whenever either or both of its inputs are true. XOR, on the other hand, is true when either of its inputs, but *not both*, are true. It is thus more closely related to what we mean when we use "or" in English: if we say "Mary or Dave is driving the car", we would be very surprised to see them both driving the car simultaneously!

The truth table and logic gate symbol for XOR are:

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0



Because of its similarity to OR, the symbol for XOR is $\oplus$. Thus, we can write it in a Boolean expression as $A \oplus B$ or A XOR B.

As we will later see, XOR is useful in constructing adders, and it should be easy to see that we can create an "equals" function by combining a NOT gate with an XOR gate. (If that's not apparent, see Sidebar **??** in the next chapter.)

Truth tables, revisited

We've seen quite a few truth tables so far, but although you may have noticed some common patterns in them, we have not yet said how to go about constructing them. We know we need all possible combinations of inputs, but we haven't yet seen how to make sure that is the case, or how to order the rows.

Let's first note that if we consider for a moment the input values themselves, then each row can be seen as a row of as binary digits (bits). Each row of inputs can then be thought of as a binary number.

As introduced in the previous chapter, a binary (base-2) number is one in which each digit is a 0 or a 1, rather than 0–9, as in decimal (base 10) numbers. We'll talk much more about binary numbers in Chapter . For now, we will just introduce them enough for our present purpose.

As with the more familiar base-10 (decimal) number system, the *least-significant digit* is on the right, while the *most-significant digit* is on the left. When we count in base 10, the least-significant digit varies fastest, then the next most-significant digit, and so on:

000, 001, 002, 003, . . ., 009, 010, 011, . . ., 099, 100, 101, . . .

This is the same when counting in binary, except the only digits we have to work with are binary digits (bits), which only have two possible values, 0 and 1 (unlike decimal digits, which have 10 possible values). Counting in base 2, then, looks like:

000, 001, 010, 011, 100, 101, 110, 111

Note that the least-significant digit changes fastest, as before.

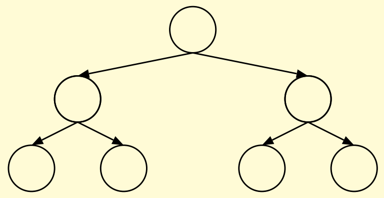
Sidebar 3: Exponential growth

It is important to realize how rapidly something increases when growing exponentially. While it may not seem very fast—for example, with 6 inputs, we still need only 64 rows for a truth table—after a little bit, exponential growth yields some very impressive numbers. If a truth table has 10 inputs, $2^{10} = 1024$ rows are needed; with 20 inputs, $2^{20} = 1,048,576$ rows. With 80 inputs—a very large circuit, we must admit—we would something on the order of a row for every atom in a mole of an element: a row for every atom in, for example, a 60 carat diamond (about one and a third times the size of the Hope Diamond), or in about 22 liters of a gas.

For an example from the physical world, suppose you make a deal with your boss that you will be paid only an atom of gold for the first month, doubling the amount each month. After a year, you would be rather poor, of course, with only 4096 gold atoms for your 12th month—so you had better have some other source of income! After even five years, your monthly income would be up to a barely-measurable third of a milligram. When you begin year seven, you would be paid about 3 g of gold your first month, worth about $3000 at the time of this writing; the year would end *much* better, though: the last month of that year, you would be paid about 6 kg of gold—worth about $6 million. Pretty good, eh? But: If you were to work for $13\frac{3}{4}$ years, then your boss would owe you a chunk of gold that weighed about a third more than the *Earth*. After a little over 15 years, he would owe you one as massive as the sun.

Why is understanding exponential growth important for computer science? One common place it occurs is when analyzing how long a program would take to solve some problem. The running time, for interesting and useful problems, often grows exponentially as the number of inputs increases. Some problems also take exponentially more memory as the size of their input increases.

To see how exponential growth in running time can impact a program's performance, consider a simple, hypothetical game in which at any point, there are two possible moves, and at some point, there is a winner. If we diagram the possible moves, we end up after two moves with something like the *tree* shown here:



That is, the number of possible resulting boards after two moves is 4, or 2^2. The tree of possible moves grows exponentially; after n moves, the number of possible boards produced is 2^n.

Suppose we want to have a game-playing program examine all possible moves until it finds a winning board position. If the winning board is only 20 moves into the game, there will be $2^{20} = 1,048,576$

boards at the "leaves" of the tree. The total number of boards that will have to be looked at will be $1 + 2 + 3 + 4 + ... + 1,048,576 = 2^0 + 2^1 + 2^2 + ... + 2^{20}$. It turns out that $2^0 + 2^1 + 2^2 + ... + 2^{n-1} + 2^n$ is equal to $2^n - 1$, so the total we have to examine for 20 levels is $2^{21} - 1$. More generally, to look for a goal at depth n in the tree, we have to look at most at $2^{n+1} - 1$ boards.

While a million or so boards might seem like a lot, if we can examine even one board per microsecond—which is not unreasonable—then it only takes around a second to look for an answer. However, if the winning board is reached only after 40 levels, then it would take the computer just over a million seconds to look for a goal. That is, about 13 days. If the game takes 80 moves to produce a win—possible in games such as chess, for example, which also has more moves to consider at each choice point—then it will take our computer something like 38 billion years to search the tree for a win; for comparison, the universe has been around for "only" something like 13.8 billion years.

A	B	C	F
0	0	0	x_0
0	0	1	x_1
0	1	0	x_2
0	1	1	x_3
1	0	0	x_4
1	0	1	x_5
1	1	0	x_6
1	1	1	x_7

Now let's consider a truth table with three inputs, A, B, and C, and one output, F. The convention is that we consider A, B, and C as binary digits comprising a single binary and number order the rows by increasing value of that number, as shown in the margin.

We do not specify values for F, since we're not concerned with them here. Note the patterns as you read down the columns; this should give you a good sense of what we mean by less-significant digits varying faster than more-significant digits. Note also that every possible combination of values for A, B, and C is represented as a row in the truth table.

The number of rows in a truth table depends only on the number of input variables. This can be understood by realizing that there has to be as many rows as there are combinations of values of the input variables. For a single variable, there need only be two rows, since there are only two values the variable can have (see the truth table for `NOT`, e.g.). For two variables, for each value of one of the variables, the other variable can also have one of two possible values; thus, there need to be $2 \times 2 = 4$ rows (see the truth tables for `AND`, `OR`, etc.). For three variables, we have 8 rows, as the truth table above shows.

Thus for every additional variable, the size of the truth table doubles. If we have n variables, then since each variable can take on one of two values, we have 2^n combinations, and thus need 2^n rows.

This is an example of exponential growth, which we will see a lot in computer science. Exponential growth makes numbers get surprisingly big very quickly. (See Sidebar 3.) This makes creating truth tables for many variables impractical. For example, with 10 variables, we need $2^{10} = 1024$ rows. For 20 inputs, we need $2^{20} = 1,048,576$ rows, and for 30 inputs, we would need over a *billion* rows.

Abstraction

One thing that we should mention before we finish the chapter is the very important idea of abstraction. Abstraction is used to hide details that would otherwise get in the way. We say that the details have been "abstracted away". We have seen an example of this already: we talk about gates, not transistors and resistors, in order to focus our thinking on digital logic and avoid thinking about the minutiae of electronics. Abstraction is a theme that runs throughout the book because it runs throughout computer science itself.

The usefulness of abstraction may be made clearer by two non-computer science examples. First, consider an automobile. We can think of the car in many different ways. It is a mode of transportation and can be thought of solely in those terms, say when we are planning a trip. It is also a large physical object and can be thought of as such when we are considering whether or not it will fit in a parking space. It is a collection of systems—the electrical system, the engine, the brakes, etc; we think of the car in these terms when something goes wrong, and this is the abstraction level that mechanics use. Each of these systems is itself made up of parts, too, and we may have cause to think of them. We could carry this process down to the level of atoms or even subatomic particles, although these low-level abstraction levels would not likely be of much use on a day-to-day basis!

The second example is the way a doctor might view a person. At the highest level of abstraction, the doctor might view the patient as a whole person with some set of complaints (e.g., pains, a cough, etc.). Going a level down in abstraction, the doctor can think about the body's systems, such as the respiratory, gastrointestinal, or circulatory systems. Each of these is composed of parts (organs), and the doctor may have cause to think of the patient at this level of abstraction: the stomach, intestines, liver, and so forth, instead of the GI system. Each organ is composed of tissues, and the doctor might need to think of the patient at that level, too: if the patient is diabetic, for example, the doctor might have to think of the patient's pancreas islet cell tissue (or lack thereof). There may be cause to go into even more detail, perhaps when thinking about some genetic disease (caused, perhaps, by a mutation in the DNA in the cells) or by some other malfunction of a macromolecule. Again, going further may not be reasonable, since most things in medicine occur above the level of individual atoms.

Why do we think of cars and bodies at various levels of abstraction? Think how hard it would be to keep all the details in mind if we were to think about a car as a collection of individual parts, much less as a collection of atoms. Not only that, but even if we could keep all the details in mind, it would be difficult if not impossible to focus on what is important for the task at hand. In addition, there are *systems properties* that are not attributable to any particular component, but that arise from the components' properties and interactions in the aggregate. There is no "digestion" ascribable to the stomach, small intestine, etc., but rather it is a property of the GI system as a whole; stopping is not a property of any particular piece of the brake system of a car, but instead is a property of all the parts working together.

In computer science, we need to think about computers at many different levels of abstraction at different times. Computers are incredibly complex machines, and it would be extraordinarily difficult always to think about a PC, for example, as a collection of gates. Software is similarly complex, and so we do not usually think of a program as its component 1s and 0s, even though that is what it is at one level.

We will see the various abstraction levels involved in computers throughout the book. Here, we just introduce them to provide an overview and a hint at what lies ahead.

The lowest level we need to be concerned with in computers is very low indeed: the quantum, or subatomic particle, level. The information stored on disk drives, for example, is becoming so densely packed that quantum effects have to be considered, even where quantum mechanics is not needed to design the storage mechanism itself. Luckily, in computer science, we seldom find ourselves at this level.

We can call the next level the electronic level. Here, we are concerned with voltages and currents. The components at this level are resistors, capacitors, coils, batteries or other voltage sources, and, of course, the ubiquitous transistors. If we understand these electronic devices, we do not have to worry about quantum mechanics, the materials the devices are constructed from, or the way their components (if any) are connected. Instead, we can treat them as black boxes with well-defined inputs, outputs, and other properties, and we can hook them together using just this information to construct what we need.

Gates, the next level up, are composed of these electronic components. However, if we understand the gate's relationship between its inputs and outputs, then we can ignore how they are constructed and concentrate on their function—that is, we can treat gates, too, as black boxes, thus hiding the details of their electronic components—as we have done in this chapter.

In computers, gates are combined to create *functional units*, such as adders (Chapter), registers (Chapter), memory, and control units. This level of abstraction is closer to the kinds of things we need to think about when designing and using a computer than gates themselves, and we can treat functional units, too, as black boxes that can be used and interconnected as desired.

Functional units are combined to create larger modules, such as the computer's arithmetic logic unit (ALU), its control unit, its register unit, etc. (see Chapter). These larger functional units give us more capable black boxes from which we can build or think about computers.

The computer is comprised of units at the next level: the central processing unit (CPU), memory, and input/output devices.

Computer scientists similarly use abstraction when thinking about or designing software. Levels here might start with the *firmware* or *microcode* that controls the internal operation of the CPU, but often this is considered part of the hardware. Software pretty much starts at the machine code level, which provides the instructions the computer is capable of carrying out. The assembly language level is next, which is basically a more human-friendly version of the machine language level. Above this might be the view a user program has of the machine and its operating system. User programs are (or should be!) themselves composed of objects or modules, which may be hierarchically organized into larger and larger modules.

We will call your attention to abstraction again and again throughout the book.

Summary

In this chapter, we have introduced digital logic, truth tables, and gates. We have seen the correspondence between gates and Boolean operators, and we have seen how to specify the behavior of an operator/gate using a truth table.

In the rest of this part of the book, we will look at digital logic in much more detail. First, we will delve more deeply into Boolean algebra, which is a way of specifying the function we would like a digital logic circuit to perform. We will see ways to prove that one function/circuit is functionally equivalent to another, which is something necessary if we are to minimize the number of gates and inputs a circuit has.

After that, we turn our attention to digital logic circuits themselves and see how we can create a circuit based on either a Boolean algebra function or a truth table. Then we will see one particular way to minimize circuits that is much easier than using Boolean algebra directly.

The final two chapters in this part of the book will look at the two types of logic circuits, combinational and sequential. For each, we will see an important example for computers (adders and registers, respectively), which will be good preparation for the next part of the book on computer architecture.

Review

Further reading

Exercises

1. Make a table showing how many rows a truth table will have for i inputs, where i goes from 1 to 10.
2. The Boolean function $\overline{A \oplus B}$ is equivalent to $A = B$. Draw the truth table for the function and argue that this is indeed the case.
3. A state-of-the-art 2012 CPU chip, the Intel Core i7, has over 1300 inputs. If we assume that each of these is a logic input (i.e., has two values), could we realistically create a truth table for this chip? Explain your answer.
4. How would you use only $\wedge$, $\vee$, and *not* to compute $A \oplus B$?
5. Show how the operators $\wedge$, $\vee$, and *not* can be computed using only *nand* gates.
6. (Difficult.) We showed in Sidebar 1 how a *nand* gate could be implemented using two transistors. Can you design a *nor* gate, also using only two transistors?