Chapter 1

Digital Logic

1.1 Introduction

Digital circuits are electronic circuits designed to operate with a fixed number of discrete voltage values. Usually—and throughout this book—where are just two such values. Depending on how we use them, we may designate them with different symbols, such as:

- *Truth values*, *true* and *false*, denoted T and F, repectively.
- *Boolean values*, 1 and 0.
- *Voltage*, the electrical potential between a point in the circuit and a common reference point, called "ground." We call these values "high" and "low," denoted H and L, respectively[1]

Digital systems record 1 and 0 in several ways:

(a) *CD*s and *DVD*s represent 1 by the *presence* and 0 by the *absence* of a pit (or depression) in a layer of aluminum depositied on a rigid plastic disk and detected by reflection of a laser beam. The same principle—presence or absence of a hole detected by a light sensor—was used in computer punch cards and paper tapes in early computers (and even before that).

(b) *Magnetic Disks* (and Data Storage Tapes) represent logic data with magnetized areas on an iron oxide recording surface. A south pole sticking out of the surface would represent a 1 and a north pole would be a 0 (or vice versa).

(c) Spring loaded *mechanical switches* have two states, *closed* and *open.* Either of these states could be chosen to represent truth; for example, the designer might interpret an open switch as a logic 1.

[1]The same voltage levels are used for signal values and power levels. For the latter, we use the names the names "power" and "ground" with symbols VCC and VDD, respectively.

(d) As already mentioned, *voltage* in digital electronic circuits. The 74LS family of transistor-transistor logic (TTL) integrated circuits, pervasive in the 1980s, uses two voltage ranges, *high voltage* (H) being 2.2–5.0 V and *low voltage* (L) being 0–0.8 V. The designer may choose to identify either level with 0 and the other with 1, as we shall see in Chapter 2. All circuits are designed to operate correctly within these ranges. In the 1990s, as power consumption became a dominating factor in portable electronics, new families of integrated circuitry emerged with high voltages ranges up to 3.2 V. The important feature remains that the circuity must be able to distinguish the two ranges.

This two-valued, or binary, characteristic of the digital world makes boolean algebra the appropriate way to mathematically model the behavior of these physical devices. Conversely, the desire to implement logical constructs in physical devices makes binary devices useful. If the devices produced more than two values, more complicated mathematics would be needed to reflect this property abstractly. In a way it is fortunate that engineers have met with only limited success in building reliable non-binary devices. Three-valued or five-valued devices would be far more difficult to trouble-shoot. Knowing that any signal must hold either a 1 or a 0 turns out to be important in practice.

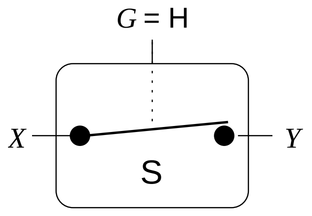
Of course, electronics is continuous by nature, and hence is infinite-valued. Since we are discussing why discrete-valued devices have only two values, it is worth contemplating why digital circuits arise in the first place. While there may be no self-contained reason for this development, it may be enough to say that they exist because we can do so much with them. Digital systems can be systematically designed that exhibit far greater functionality than has ever been attained using *analog* electronics.

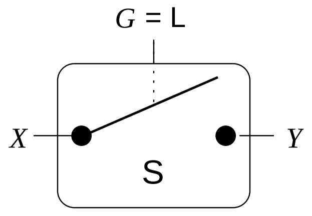
Again, why?

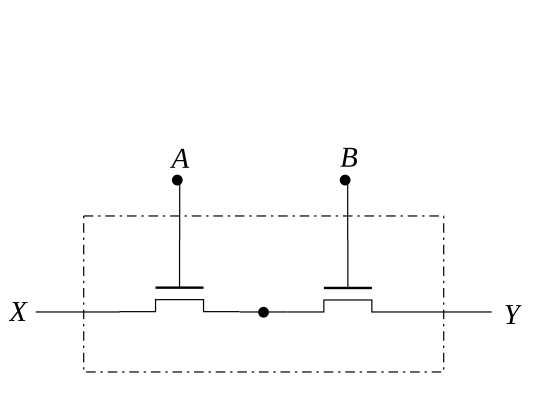
One important reason is that the *functionality* of active logic devices orients them and restricts their behavior with respect to input and output. In contrast, analog electronics are *relational* in character and hence more difficult to manipulate mathematically. These devices are also *temporally* oriented, an aspect to be much further discussed in later chapters of this textbook.

1.2 Digital Devices

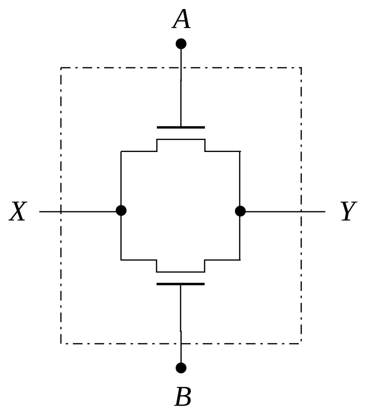
The electronically controlled *switch*: is a fundamental digital device.







(a) Connected in series

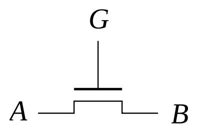


(a) Connected in parallel

Figure 1.1: Two ways to combine two switches to connect points X and Y

A high voltage at the point G *closes* this switch, making a connection between points X and Y. A low voltage at G *opens* the switch, disconnecting X and Y.

Switches integrated circuits are realized with *transistors*, depicted in schematics by this symbol:

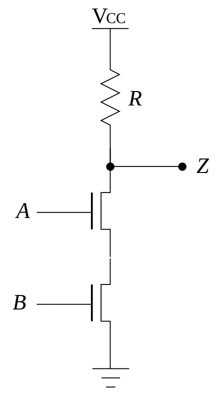


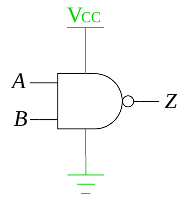
Chapter **??** describes how transistors work.

The logical function of a circuit derives from the elementary ways we can combine switches. Fig. 1.1 shows the two ways to combine switches S_1 and S_2 between two connecting points, X and Y. In Fig. 1.1(a), on the left, they are connected in *series*; and in Fig. 1.1(b), on the right, they are connected in *parallel.* If S_1 and S_2 are connected in *series*, then there must be high voltage at *both point A and point B* to have a connection between points X and Y. Thus, in this sense the series connection represents the function and. Similarly, If S_1 and S_2 are connected in *parallel*, then X and Y are connected when there is high voltage at *either point A or point B, or both*, so the parallel connection represents the logical function or.

In theory, we could use this idea to implement any logical expression of *and*s and *or*s. However, our idealized concept of a perfect switch cannot be realized by physical devices, and we must do a bit more to sustain the illusion of "digital logic." Both the impulse needed to open or close the switch and the imperfect (dis)connection it makes consume electrical energy. The most significant consequence of this energy drain is to compromise the discrete voltage levels on which the logic is based in the first place. If a high voltage closes the switch and a low voltage opens it, an "intermediate" voltage leaves the state of the switch uncertain. It takes only a few real switches to create such an ambiguous situation.

Real digital components are *active devices* that employ *amplification* to restore signals to their proper digital levels. Below, two transistors are connected in series with a resistor (an electronic element that restricts current flow), between VDD ("ground") and VCC ("power"), both constant voltage sources.





Now suppose that high voltage is present at both points A and B, so that both switches are closed, as before. The point Z is thereby more directly (i.e. with less resistence[2]) connected to ground through the switches; it therefore acquires a low voltage. On the other hand, if either of the switches is open, Z is disconnected from ground and the connection to VCC dominates.

In either case, the voltage at point Z is restored to a proper digital level and, in addition, this voltage has the full strength of the power supply behind it. Hence, the signal Z is clear and strong.

This device is an example of a digital *logic gate*, and there are several points worth remembering about it.

- As has already been noted, it is an *active* device, using amplification to sustain proper voltage levels for digital operation.

- The gate device is behaving as a "function" with respect to *inputs* A and B and *output* Z. One schematic symbol, called nand, for that function is

[2]The voltage at Z is $(\text{Vcc} \cdot R)/(R + r_s)$, where r_s is the resistance across the two switches. R is chosen to minimize leakage when the switches are closed and maximize current when the switches are open. This figure illustrates NMOS technology. We will look at CMOS technology (in which the resister is replace by another transistor) later.

NOTE: This is not a course in electronics. This discussion, and others like it throughout the text, describe how digital components work in a qualitative way. The purpose is to provide intuition about the physical basis of digital systems. However, there is not ground material in this textbook to understand how or why these physical objects behave the way they do. Your instructor may choose to provide additional material about this level, or direct you to other courses to learn more. In any case, this groundwork is not critical for understanding the main topics developed in this book.

shown to the right of the schematic above. We might read it as saying "$Z = A$ *nand* B."

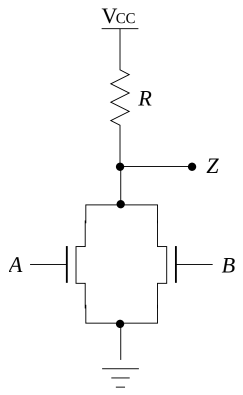
- Because it uses amplification, the gate requires a power supply for correct operation. It also generates physical byproducts, such as heat. Its ability to sustain our idealized view of its operation depends critically on our using it properly, providing adequate power and heat dissapation, and so forth.

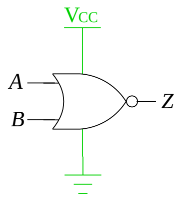
- Because it is a physical device the gate takes time to perform its function. This crucial fact will become the focus of our attention starting in Chapter **??** and onward. Generally speaking, the temporal aspect of system design poses the most daunting intellectual challenge to designers and therefore becomes central to our design methodology.

One of the first things we do in developing a disciplined digital design method is abstract from (or remove from consideration) details that get in the way of our paramont design objective: achieving some specified functionality. During design, we *want* to think of our building blocks as purely mathematical functions, rather than physical devices, to do otherwise merely clouds our thinking. When a *nand* gate appears in a schematic, the power and ground connections are not shown because they are irrelevant to the principal design goal: the function of the design. To show them would add needless clutter and distraction.

Of course, when the time comes to build the physical implementation, the engineer had better remember to connect the power supply! This is a theme that we shall revisit again and again in this book. The concepts and methods for design have co-evolved over time with the technolgies and building blocks available for implementation. Understanding the juxtaposition of the conceptual "design space" and its reflection in physical building blocks is the key to making sense a method and to keeping up with unrelenting advances in technology and practice.

If the series-connected transistors in the *nand*-gate are replaced by parallel connected transistors, as below, the result is a logic gate whose output is grounded when the voltage on either of its inputs is high.





This is traditionally called a *nor* gate. We shall see in the next chapter that the names "nand" and "nor" are misleading.

1.3 Logic and Boolean Algebra

1.3.1 Logic Constants and Primitive Operations.

The logical values *true* (T) and *false* (F) have already been introduced. Our use of logic in this chapter is restricted to propositional expression, involving simple combinations of logic values, without quantification (i.e. without "for all," "there exists," etc.). Of particular interest are the functions *and* (denoted by an infix '$\wedge$'), *or* (denoted by an infix '$\vee$'), and *not* (denoted by a prefix '$\neg$'). In logical expressions, the unary "*lnot*' has highest precedence and '$\vee$' lowest, so for example, the term

$$S \wedge A \vee \neg S \wedge B$$

if fully parenthesized, would be written

$$(S \wedge A) \vee ((\neg S) \wedge B)$$

(See Exercise 3.)

1.3.2 Logical Variables.

Capitalized italic names A, B, C, etc. are used for logic variables. We may employ subscripts in groups of related names; for instance, A_0, A_1, $\ldots A_7$.

In design examples, nmemonics like *LD*, *CLR*, are used and, when the situation calls for it, descriptive identifiers, such as *PHOTODIODE.ERROR*, *START.ASSERTED*. A period is used as a mnemonic separator in this book, but different design tools and other sources may use hyphens or underscores for this purpose.

1.3.3 Truth Tables

A *truth table* is a representation of a logical function. The first example, on the left below, describes a function X of three variables, A, B and C. The second example, on the right, describes a two functions, Y and Z, of variables S A and B.

A	B	C	X
F	F	F	F
F	F	T	T
F	T	F	T
F	T	T	F
T	F	F	T
T	F	T	F
T	T	F	F
T	T	T	F

S	A	B	Y	Z
F	F	T	F	T
T	F	F	F	F
F	T	F	T	T
T	F	T	T	F
F	F	F	F	T
T	T	F	F	F
T	T	T	T	F
F	T	T	T	T

$$X(A, B, C) = [\mathsf{F}, \mathsf{T}, \mathsf{T}, \mathsf{F}, \mathsf{T}, \mathsf{F}, \mathsf{F}, \mathsf{F}]$$

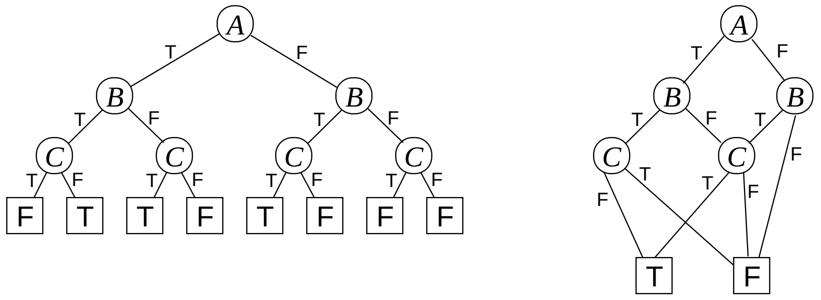


Figure 1.2: Some truth table representations: an array (top), a decision tree (left), and a decision diagram (right).

If you study the truth tables for X, Y and Z long enough, you may probably think of better ways to describe them, either informally with English predicates or perhaps formally as logic expressions.

You may also have noticed a difference in the order that the rows of these two tables are presented. Although the order does not matter, so long as there is just one row for each of the eight possible input combinations, it is generally better to have a standard, or *canonical*, ordering. For reasons that are explained in Section 1.3.5, the ordering on the left, above, is canonical.

If we are going to use the standard ordering, then the only thing we really need to describe a particular truth table are the function's name, its variables in some order, and the column of specified function values. The rest is "boiler-plate," giving the table a familiar shape and format. Figure 1.2 shows a number of possible representations this essential information might have, including data structures that often used for computer representations. Of particular note is the *binary decision diagram* (*BDD*), Fig. 1.2, lower left, a computer representation found in many design tools.

Figure 1.3 gives truth tables defining some of the primitive logic functions.

1.3.4 Elements of Boolean Algebra

In mathematics, a *boolean algebra* is one of a family of systems which share certain properties.[3] For our purposes "boolean algebra" refers to a particular

[3]The term *boolean* derives from the name of the mathematician George Boole (1815–1864) who was the first to apply the concept of algebra to logic. Some textbooks acknowledge Boole's profound contributions to computer science by using a lower-case 'b'.

not, $\neg$	
A	$\neg A$
F	T
T	F

and, $\wedge$		
A	B	$A \wedge B$
F	F	F
F	T	F
T	F	F
T	T	T

or, $\vee$		
A	B	$A \vee B$
F	F	F
F	T	T
T	F	T
T	T	T

either-or, $\neq$		
A	B	$A \neq B$
F	F	F
F	T	T
T	F	T
T	T	F

iff, $=$		
A	B	$A = B$
F	F	T
F	T	F
T	F	F
T	T	T

implies, *only if*, $\supset$		
A	B	$A \supset B$
F	F	T
F	T	T
T	F	F
T	T	T

Figure 1.3: Some primitive logic functions.

system involving two constants, 1 and 0, and operations $+$, $*$ and $-$, defined as

$-$	0
0	1
1	0

$+$	0	1
0	0	1
1	1	1

$*$	0	1
0	0	0
1	0	1

Other members of the family include

- The theory of sets, $\emptyset$ playing the role of 0, a universal set $\mathcal{U}$ playing the role of 1 and *set intersection*, *set union* and *set complementation* (with respect to $\mathcal{U}$) playing the roles of $*$, $+$ and $-$, respectively.
- Logic, with $\wedge$ for $*$, $\vee$ for $+$ and $\neg$ for $-$.
- Arcane instances, such as the system $\mathcal{B}_N$ whose values are all the divisors of a number N with 1 for 0, N for 1, *least-common-multiple* for $+$, *greatest-common-divisor* for $*$ and $\overline{x} =_{\text{def}} N/x$.

Of course, with just two constants, T and F, logic is essentially identical to the binary system just introduced. We will use these two systems interchangably, and even say "true" for 1, "and" for '$\cdot$', and so forth. In fact, the constants 0 and 1 are often to be prefered over T and F because they are easier to write down and are more visually distinct. Furthermore, the boolean operator symbols are chosen for their familiarity and hence are easier to manipulate algebraically.

Despite these advantages, we will still use the logical operators when we wish to emphasize logical aspects of a design. For instance, if we want a signal $READY$ to be asserted whenever both conditions A and B are true, we will likely write

$$READY = A \wedge B$$

(a)	$\overline{\overline{x}} = x$		*double negation*
(b)	$x \cdot 1 = x$	$x + 0 = x$	*identity*
(c)	$x \cdot 0 = 0$	$x + 1 = 1$	*dominance*
(d)	$x \cdot x = x$	$x + x = x$	*inversion*
(e)	$x \cdot \overline{x} = 0$	$x + \overline{x} = 1$	*cancellation*
(f)	$x \cdot y = y \cdot x$	$x + y = y + x$	*commutativity*
(g)	$x\,(y\,z) = (x\,y)\,z$	$x + (y + z) = (x + y) + z$	*associativity*
(h)	$x\,(y + z) = x\,y + x\,z$	$x + y\,z = (x + y)\,(x + z)$	*distributivity*
(i)	$\overline{x \cdot y} = \overline{x} + \overline{y}$	$\overline{x + y} = \overline{x} \cdot \overline{y}$	*DeMorgan's law*
(j)	$x\,(x + y) = x$	$x + x\,y = x$	*absorbtion*
(k)	$x\,(\overline{x} + y) = x\,y$	$x + \overline{x}\,y = x + y$	*absorbtion*
(l)	$x\,y + \overline{x}\,y = y$	$(x + y)\,(\overline{x} + y) = y$	*independence*

Figure 1.4: Boolean identities

to express the logical intent.

As just mentioned, the choice of symbols for the boolean *and*, *or* and *not* operations reflect the fact that their algebraic properties are similar to (but stronger than) those of arithmetic multiplication, addition and negation. We may use familiar algebraic identities, such as commutativity and associativity, to simplify and reason about boolean expressions. Figure 1.4 lists some of these identities and names some of them. Some of the identities can be derived from the others in the list. Since we mean to take advantage of our hard-learned familiarity with the laws of arithmetic, we shall adopt the same notational conventions with our boolean operators. "Times" takes precedence over "plus," and wherever possible we drop the '·' and simply juxtapose the operands of a product.

Since both '+' and '·+' are associative, we do not need to parenthesize long strings of sums or products. Furthermore, the distributive and DeMorgan laws generalize, for instance:

$$\overline{(A + B + C + D + E)} = \overline{A} \cdot \overline{B} \cdot \overline{C} \cdot \overline{D} \cdot \overline{E}$$

and

$$A \cdot (B_1 + B_2 + \cdots + B_n) = (A \cdot B_1) + (A \cdot B_2) + \cdots (A \cdot B_n) +$$

Although the boolean operators share many properties with their arithmetic counterparts, there are also some significant differences. You will see in Fig. 1.4 that all of the identities involving $+$ or $\cdot$ come in pairs. This feature reflects the

The Principle of Duality: From any valid boolean equation $\mathcal{L} = \mathcal{R}$ one may derive another valid equation $\mathcal{L}^D = \mathcal{R}^D$ by interchanging 1s with 0s and '+'s with '·'s. The expression $\mathcal{E}^D$ obtained from $\mathcal{E}$ in this way is called the *dual of* $\mathcal{E}$.

In performing boolean derivations, one often misses opportunities to apply the duals of the familiar arithmetic laws, practiced over and over in high school and calculus classes. Fortunately, one is unlikely to be called upon to perform extensive derivations in digital design applications. Nevertheless, it is useful to be able to do short "off the cuff" derivations, so a bit of practice is worthwhile.

DeMorgan's Law, which says that negation "distributes" over *both* '+' *and* '·', give another aspect to duality. DeMorgan's Law can be stated more generally than the primitive law appearing in Fig. 1.4:

DeMorgan's (Generalized) Law: The negation of any boolean expression $\mathcal{E}$ is equivalent to the expression obtained by negating all the variables within $\mathcal{E}^D$.

DeMorgan's Law makes it convenient to express negation by drawing a bar over the negated expression rather than putting a minus-sign in front of it, as we do in arithmetic expressions. However, if this convention is used carelessly in combination with other shortcuts, *there is a danger of confusing the negation of a product with the product of negatives*:

$$\overline{AB} \neq \overline{A}\,\overline{B}$$

Be careful!

The general form of DeMorgan's Law is proved valid by an induction over the grammar of logical expressions. We have not laid the groundwork for such a proof in this textbook. Let us look at an example, at least.

Example 1. According to DeMorgan's Generalized Law, it should be the case that

$$\overline{A + B\,\overline{C} + \overline{A\,C\,\overline{D} + B\,D}} = \overline{A}\cdot(\overline{B} + \overline{\overline{C}})\cdot\overline{(\overline{A} + \overline{C} + \overline{\overline{D}})\cdot(\overline{B} + \overline{D})}$$

Let us prove this instance algebraically, using the repeated applications, from left to right, of the primitive form of Fig. 1.4(i).

$$\begin{aligned}
\overline{A + B\,\overline{C} + \overline{A\,C\,\overline{D} + B\,D}} &= \overline{A + B\,\overline{C}}\cdot\overline{\overline{A\,C\,\overline{D} + B\,D}} \\
&= \overline{A}\cdot\overline{B\,\overline{C}}\cdot\overline{\overline{A\,C\,\overline{D} + B\,D}} \\
&= \overline{A}\cdot(\overline{B} + \overline{\overline{C}})\cdot\overline{\overline{A\,C\,\overline{D} + B\,D}} \\
&= \overline{A}\cdot(\overline{B} + \overline{\overline{C}})\cdot\overline{\overline{A\,C\,\overline{D}}\cdot\overline{B\,D}} \\
&= \overline{A}\cdot(\overline{B} + \overline{\overline{C}})\cdot\overline{(\overline{A} + \overline{C} + \overline{\overline{D}})\cdot\overline{B\,D}} \\
&= \overline{A}\cdot(\overline{B} + \overline{\overline{C}})\cdot\overline{(\overline{A} + \overline{C} + \overline{\overline{D}})\cdot(\overline{B} + \overline{D})}
\end{aligned}$$

So the generalized law holds in this case.

Although it sometimes provides a convenient shortcut, DeMorgan's general law is less often applied in practice than its more primitive form in Fig. 1.4(i). Often, when double-negations arise—as in the second step of the derivation above—they are immediately simpified using Eq. 1.4(a) Application of the law can be easier said than done because the precedence rules often require (or allow) us to introduce (or eliminate) parentheses when when '$\cdot$'s and '$+$'s are interchanged.

Let us work a few examples to get started.

Need some more examples

$$\begin{aligned}
\overline{A\cdot(B+C\cdot(B+\overline{A}))} &= \overline{A\cdot(B+C\cdot B+C\cdot\overline{A})} && \text{(Eq. 1.4(h))}\\
&= \overline{A\cdot(B+C\cdot\overline{A})} && \text{(1.4(j))}\\
&= \overline{A\cdot B+A\cdot C\cdot\overline{A}} && \text{(1.4(h))}\\
&= \overline{A\cdot B+A\cdot\overline{A}\cdot C} && \text{(1.4(f))}\\
&= \overline{A\cdot B+0\cdot C} && \text{(1.4(d))}\\
&= \overline{A\cdot B} && \text{(1.4(c), 1.4(b))}\\
&= \overline{A}+\overline{B} && \text{1.4(i))}
\end{aligned}$$

1.3.5 Bits, Binary Numerals

We often interpret a group of binary variables as a binary number, in which case it is customary to display these variables in such a way that the least-significant bit is right-most, as in normal numerical representations.

When an indexed group of variables, A_0, A_1, …, A_k is involved, A_i usually represents the "2^i's place" in the corresponding numeral, and we will write the $k+|$-bit numeral out as $A_k\,\cdots\,A_1\,A_0$ with the least-significant bit, again, furthest right.

An immediate application of this idea is in the presentation of truth tables. Our first example of a truth table is repeated below, with truth values on the left and the corresponding boolean values on the right

A	B	C	X
F	F	F	F
F	F	T	T
F	T	F	T
F	T	T	F
T	F	F	T
T	F	T	F
T	T	F	F
T	T	T	F

	A	B	C	X
(0)	0	0	0	0
(1)	0	0	1	1
(2)	0	1	0	1
(3)	0	1	1	0
(4)	1	0	0	1
(5)	1	0	1	0
(6)	1	1	0	0
(7)	1	1	1	0

This truth table is in standard form because its rows are listed in numerical order, depending a fixed choice of first A, then B, and finally C as the ordering of its input variables. In this case, we have (redundantly) numbered the rows for reference.

We shall have a great deal more to say about the binary numeric representations and the implementation of arithmetic in later chapters.

1.4 Summary

This chapter has set the scene for the topics that follow. Like most chapters in this textbook, this one has looked at both physical and conceptual aspects of digital systems. As in common in most textbooks about "design," we are concerned with resolving two levels of abstraction. Of course, we are just at the outset of this process, so the distinction is rather course. We have taken a brief, qualitative glimpse at what digital systems—a vast domain ranging from wristwatches to global communication systems—are made of: switches. And we have taken the first step toward imposing a useful conceptual function on those elementary devices: logic gates.

We want to get started on a good footing. In the next chapter, we will explore a vital, fundamental principle of design in detail, before returning to the task of expanding our design methodology.

Exercises 1.4

1. Figure 1.3 defines five binary (two-input) logic functions. How many two-input logic functions are there?

2. On Page 4 a *nand* gate is described by an "equation," $Z = A \textit{ nand } B$. Discuss what the '=' sign might represent in this context? Is it mathematical equality? A defining expression? An assignment statement?

3. Verify that the truth table for Y on page 7 specifies the function

$$Y = S \wedge A \vee \neg S \wedge B$$

4. By inserting full parentheses, show the order of evaluation of these function.

(a) $B \cdot \overline{A \cdot C} + D + \overline{E}$

(b) $\overline{A + B} \cdot C + D$

(c) $\overline{A + B} \cdot (\overline{C} + D)$

5. By inspecting their truth tables, deduce the value of each function, X, Y, Z and W. Do not formally derive any logic expression; simply write down the minimal formulation of each function.

A	B	C	W	X	Y	Z
0	0	0	0	1	0	1
0	0	1	0	1	0	0
0	1	0	0	1	0	1
0	1	1	0	1	0	0
1	0	0	0	1	1	1
1	0	1	0	1	1	0
1	1	0	0	1	1	1
1	1	1	0	1	1	0

6. Give 5-bit binary representations for the decimal numbers 5, 21, 48 and 13.

7. What numerical values are represented by the binary numbers 1011, 11011, 011 and 1111111111?

8. Describe algorithms for adding and subtracting binary number representations. Implement and test these algorithms in software using the language of your choice.