Chapter 1

Spring 2010 Edition

Digital Circuits, Binary Numbers and Truth Tables

In this book you will learn how to design and build real digital circuits (hardware) such as a (digital) *adder*, i.e., a digital circuit which takes two operands (*added* and *addend*), adds them up, and then sends out the *result* (output), i.e., result = added + addend. We will refer to the adder example throughout this chapter to exemplify the basic concepts introduced here. It goes without saying that this book is a real prerequisite if you wish to thoroughly understand the hardware design of revolutionary digital systems such as microprocessors[1], which are ever-increasingly and inevitably entering our daily lives.

What is a *digital* circuit?

In digital (as opposed to analog) circuits, inputs and outputs (and also intermediate signals) may only take discrete (integer) values. In other words, the number of different values that may be applied to a digital circuit, and also the number of different values that may be produced and outputted by such a circuit are limited. For example, consider a small digital adder in which added and addend each may take integer values 0, 1, 2 or 3; and therefore the output of the adder may take integer values 0, 1, 2, 3, 4, 5 or 6. Compare this digital adder[2] with an analog adder. An analog adder is an analog circuit that takes two input values (such as 1.8634… and 0.3901) from a *continuous* range of voltage, and generates an output value (such as 2.2535…), which again belongs to a *continuous* range of voltage.

Integer values used in digital systems are *binary* (radix 2 or base 2) due to the *convenience* of manufacture and operation *reliability* offered by binary hardware. Soon, binary numbers will be introduced in this chapter.

Why *digital* circuits?

The following are some basic reasons why one might prefer digital to analog. But they may not sound very clear to you if you do not have sufficient background in analog circuits. Do not worry! This will not prevent you from reading and understanding the rest of this chapter.

Reproducibility of results: A properly designed digital circuit always produces the same output for the same input. However, factors such as part replacement, temperature, aging and power-supply voltage may affect the response of an analog circuit.

Programmability, hence flexibility: Microprocessors, the most well-known digital systems, are programmable, i.e., the same hardware is able to do a broad range of jobs, and this makes them tremendously flexible.

Higher noise immunity, hence more reliability: Digital systems are inherently more immune to environmental noise, making them more reliable. In other words, digital hardware possesses tolerance against input signal variations.

Adder example (Cont'd): Being limited to binary inputs and outputs, this adder needs two binary digits for each operand and three binary digits for the result. **Why?** In order to be able to answer this question we need to take a look at *binary numbers*. Number systems will be covered in more detail in Chapter 5.

---

[1] Design of complicated digital systems such as microprocessors needs substantial computer aids, which are out of the scope of this book. But remember that the techniques you learn in this book should still enable you to design simple versions of such systems, although with poor efficiency.

[2] For this add function we may formally write: domain = (i, j), where $0 \le i$ and $j \le 3$, and range = $\{y \mid 0 \le y \le 6\}$

Binary Numbers

The traditional radix-10 or decimal system, which we use everyday, is a *positional* number system because each digit's *position* has its own specific *weight*. The weighted sum of the digits is the value of the number in hand. For example

$$2736 = 10^3 \times 2 + 10^2 \times 7 + 10^1 \times 3 + 10^0 \times 6 = 1000 \times 2 + 100 \times 7 + 10 \times 3 + 1 \times 6$$

The four positions of digits 2, 7, 3 and 6 have weights equal to 10^3, 10^2, 10^1 and 10^0, respectively, in which 3, 2, 1 and 0 represent the positions of these digits, respectively.

The same idea can easily be adapted for radix-2 or binary numbers, which are comprised of only two different binary digits, 0 and 1. "Binary digit" is abbreviated as *bit*. For example

$$101101 = 2^5 \times 1 + 2^4 \times 0 + 2^3 \times 1 + 2^2 \times 1 + 2^1 \times 0 + 2^0 \times 1 = (45)_{ten}$$

The six bit positions (from left to right) in this binary number have weights equal to 2^5, 2^4, 2^3 , 2^2 , 2^1 and 2^0 , respectively, in which 5, 4, 3, 2, 1 and 0 represent the positions of the corresponding bits. Notice that position *zero* always belongs to the *least significant bit* or LSB for short.

Binary-to-Decimal and Decimal-to-Binary Conversions

Calculate the weighted sum of the bits of a binary number to get the number's decimal representation.

Example 1. Convert $(100101)_{two}$ to decimal.

$$100101 = 2^5 \times 1 + 2^4 \times 0 + 2^3 \times 0 + 2^2 \times 1 + 2^1 \times 0 + 2^0 \times 1 = 32 + 0 + 0 + 4 + 0 + 1 = (37)_{ten}$$

Decimal-to-binary conversion may be performed through successive divisions by 2 as explained in Example 2.

Example 2. Use the *successive division-by-2* algorithm to convert $(235)_{ten}$ to binary.

The successive divisions for this conversion are shown in Figure 1. Notice that the devisor for all the divisions is always 2. The first dividend is the number to be converted to binary (235, in this example). In Figure 1, the remainder from each division has been circled and then the corresponding quotient is shown next to the remainder. The quotient from each division becomes the dividend for the following division. When a quotient becomes 0, the conversion terminates. The first remainder is the rightmost or least significant bit (LSB) of the resulting binary number. The following remainders comprise more significant bits in ascending order. Therefore, $(235)_{ten} = (11101011)_{two}$.

$235 \div 2 = (1)\ 117 \div 2 = (1)\ 58 \div 2 = (0)\ 29 \div 2 = (1)\ 14 \div 2 = (0)\ 7 \div 2 = (1)\ 3 \div 2 = (1)\ 1 \div 2 = (1)\ 0$

Figure 1. Decimal (235) conversion to binary using successive division-by-2 algorithm

It can be shown that the range of numbers that fit in n bits is $[0 : 2^n - 1]$, i.e., 0 is the smallest and $2^n - 1$ is the largest possible n-bit number. In summary, for the n-bit binary number system we may write:

- Range of numbers, $R = [0 : 2^n - 1]$ Note: this *closed* interval specified by square brackets includes the boundary numbers 0 and $2^n - 1$.
- Smallest number, $N_{min} = 0$
- Largest number, $N_{max} = 2^n - 1$

  Note: there are $(M + 1)$ integer numbers in the closed interval $[0 : M]$, or in general there are $(M - K + 1)$ numbers in the closed interval $[K : M]$. Therefore,

- Number of all possible (integer) numbers, $B = 2^n$

As an example, all possible 4-bit numbers together with their decimal equivalents are listed in Figure 2.

Binary Number	Decimal Equivalent	Binary Number	Decimal Equivalent
0000	0	1000	8
0001	1	1001	9
0010	2	1010	10
0011	3	1011	11
0100	4	1100	12
0101	5	1101	13
0110	6	1110	14
0111	7	1111	15

Figure 2. 16 possible 4-bit numbers and their decimal equivalents

It is obvious that any n-bit string (or pattern) represents a unique number, and also any number can be represented by a unique n-bit pattern (provided that the number is not too large to fit in n bits; otherwise it is not representable at all). Therefore,

- Number of all different n-bit patterns, $P = 2^n$ (1)

Example 3. How many numbers are there in the interval [0:145]?

# of numbers = 145 + 1 = 146

Example 4. How many numbers are there in the interval [29:84]?

# of numbers = 84 – 29 + 1 = 56

Example 5. What is the largest 6-bit number?

$N_{max} = 2^6 - 1 = (63)_{ten} = (111111)_{two}$

Example 6. What is the smallest 5-bit number?

$N_{min} = (00000)_{two}$

Example 7. How many 5-bit numbers are there?

$B = 2^5 = 32$

Example 8. How many 5-bit patterns are there?

$P = 2^5 = 32$

Example 9. How many bits (n) do we need to represent the decimal numbers in the range of [0:15]?

This is the reverse of Example 8. We need 15 + 1 = 16 different bit patterns to allocate to 16 decimal numbers or in general to 16 different *choices*. In the context of equation (1), now P = 16, and we have been asked to determine n. Equation (1) has been shown again here for ease of reference:

$P = 2^n$ (1)

Considering (1), n may be expressed as a function of P:

$n = \log_2 P$ (2)

Substitute 16 for P to obtain n:

$n = \log_2 (16) = 4$ bits (remember that $2^4 = 16$)

Example 10. How many bits (n) do we need to represent the decimal digits, [0 : 9]?

Number of choices, P = 9 + 1 = ten

Substitute ten for P in (2)

$n = \log_2 ten$

One way to manually obtain $\log_2 ten$ (or log of any small number) is *trial and error* after we take a reasonable initial guess. Let us initially guess that n = 5. Substitute 5 for n in (1):

$P = 2^5 = 32$

Notice that 32 > ten, so let us try n = 4:

$P = 2^4 = 16$

16 is still greater than ten, so let us try n = 3:

$P = 2^3 = 8$

8 is less than ten; therefore $\log_2 ten$ is between 3 and 4. Using a calculator one would obtain $\log_2 ten$ = 3.321 … In this example three bits are not sufficient; therefore, we choose 4, *the smallest integer that is greater than 3.321….*

A different approach: It can be proved that the number of bits that we need to represent P choices is equal to the number of bits that we need to represent number P – 1. Therefore, in Example 10 we could say:

Ten – 1 = 9; but 9 = 1001. So, we need four bits to represent ten choices, or more specifically, the decimal digits, [0 : 9]. ◊

Notice that in each of Example 9 and Example 10 we need 4 bits, although there are different numbers of choices in these two examples. In Example 9 all possible 4-bit patterns (16 in total) are needed, while in Example 10 only 10 out of 16 possible bit patterns are really used. More specifically, in Example 10 the bit patterns 1010, 1011, 1100, 1101, 1110 and 1111 are not needed; hence are left unused. So, *bit utilization* in this example (unlike Example 9) is not 100%.

Similar inefficient use of resources can be made in more familiar scenarios as well. Suppose that one bedroom is needed for every two family members. Now the question is "What size home does a 3-member family need?" And the answer is "two-bedroom". Notice that a two-bedroom home is larger than what is needed, but one-bedroom home is too small; so a two-bedroom home has to be chosen.

Conclusion: In mapping of distinct bit patterns to different choices, and to make use of bits in the most efficient way, the number of choices has to be a power of "2". This results in no unused bit patterns.

Example 11. There are 59 rooms in a building, and we are to assign one unique binary number (starting with 0) to each room. How many bits do we need? Stated slightly differently, how many bits (n) do we need to specify 59 different choices?

59 – 1 = 58; but 58 = 111 010. So, we need 6 bits.

We may interpret the unique bit pattern assigned to each room as the *address* of that room because each bit pattern in the range of 000000 (= 0) to 111010 (= 58) unambiguously locates exactly one room.

Example 12. How many rooms (maximum) can be addressed with 8 bits?

This question is similar to Example 8, hence may be put into the same familiar format: How many different 8-bit patterns are there?

n = 8, use (1) to obtain P:

$P = 2^8 = 256$.

Adder Example (Cont'd): Remember that each input is supposed to be in the range of [0:3], i.e., the number of choices for each input is $3 + 1 = 4$. Therefore, each input needs to be $\log_2(4) = 2$ bits wide with 100% efficiency because there is no unused bit pattern at either of the two inputs. (2 is also called the *size* of this adder.) On the other hand, the output of this adder may go up to $3 + 3 = 6$. Since $\log_2(6 + 1) = 2.807\ldots$, the output has to be 3 bits wide. But notice that bit pattern 111 that corresponds to number 7 may never appear at the output because the sum of two 2-bit numbers may never exceed 6, which means that the output bits are not fully efficiently used. Figure 3*a* shows a symbol for this 2-bit adder.

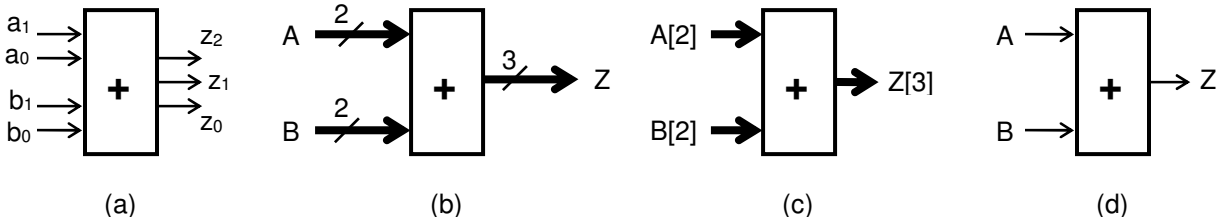

Figure 3. (a) Symbol for a 2-bit adder, (b) and (c) shorthand symbols for a 2-bit adder, (d) symbol for an analog adder

$A = a_1\ a_0$ and $B = b_1\ b_0$ are the two 2-bit input vectors (operands), and $Z = z_2\ z_1\ z_0$ is the 3-bit output vector (result) of this adder. The bit with the highest index (a_1, b_1 or z_2 in this example) is called the *most significant bit* (or MSB for short) in the corresponding bit vector. Similarly, the bit with the zero index (a_0, b_0 or z_0 in this example) in a bit vector is called the least significant bit or LSB for short, as previously stated. By an input (or output) *bit vector* (or simply *vector*) we mean an input (or output) that is wider than one bit. Also, by $Y = y_1\ y_0$ we mean a 2-bit vector (Y) that is obtained by concatenating 2 single bits, y_1 and y_0, exactly in the same order. For example, if $y_1 = 0$ and $y_0 = 1$ then $Y = 01$. This notation can easily be generalized to longer vectors. Two self-explanatory shorthand symbols for the same adder are shown in Figure 3*b* and Figure 3*c*.

Figure 3*d* shows a symbol for an analog adder. Remember that unlike a digital adder each operand (A or B) now has only one (analog) input line driving the adder. The output (Z) is also placed on a single line.

Question: Why do we not use one *single* input line for A (instead of two lines, a_1 and a_0) and one *single* line for B (instead of two lines, b_1 and b_0) to apply all possible discrete values (0, 1, 2 and 3) to a two-bit digital adder?

Answer: If we do this, then it means that one single (input) line may take on four different values (namely 0, 1, 2 or 3). But this is a violation of the restriction that each node in a digital system may only take two values (0 or 1). This restriction is the cost that we pay to make the hardware building blocks and eventually digital systems simpler and more reliable. ◊

Before we move on, let us clarify some terminology-related issues:

We use the terms "logic circuit", "digital circuit" and "digital logic circuit" interchangeably.

"Input", "input line" and "input value": Every digital circuit has one or more input variables or input *lines* (wires). In our adder example, there are four input lines, namely, a_1, a_0, b_1 and b_0. These lines take an *input value* or *input combination* (such as 1011) at any given time. The term *input* may be used for two different purposes: *input line(s)* or *input value* (*input combination*). The intention should be clear from the context. The same is true for "output", "output line" and "output value".

The terms *variable* and *signal* might be used interchangeably unless otherwise specified. We normally use the term *variable* in truth tables and also in the algebraic domain, as you will see in Chapter 3. In the context of physical devices and circuits this term is usually replaced with the term *signal* (see Chapter 2).

When we say a line is *asserted* we mean that it is at logic 1. A line at logic 0 is called *deasserted*. This definition will be extended in Chapter 3.

Truth Tables: Tabular Description of Functions

Adder Example (Cont'd): Figure 4*a* shows the traditional decimal addition table for our tiny adder. The table has an input column comprised of the two operands, added (A) and addend (B), and an output or sum column, Z. All possible combinations of A and B (16 in total) have been listed in the input column in ascending order; the output column shows the sum of the two operands.

The addition table shown in Figure 4*b* is the binary equivalent for the decimal table of Figure 4*a*. Besides one input and one output column, there is a decimal row number for each row in this binary table. The input column of this table is 4 bits wide because A (= a_1a_0) and B (= b_1b_0) each are 2 bits wide. All the 16 possible input combinations have been placed in the input column in ascending order (see also Figure 2). Remember that using four bits we can create up to 16 (= 2^4) different bit patterns. Take a second look at the row numbers: The row number in each row is the decimal equivalent for the binary number sitting in the input column of that row. The output or sum column, Z (= $z_2z_1z_0$), of the table in Figure 4*b* is three bits wide (see also Figure 3*a*).

A	B	Z
Added	Addend	Sum
Input		Output
0	0	0
0	1	1
0	2	2
0	3	3
1	0	1
1	1	2
1	2	3
1	3	4
2	0	2
2	1	3
2	2	4
2	3	5
3	0	3
3	1	4
3	2	5
3	3	6

(a)

	A	B	Z
Row	a_1 a_0	b_1 b_0	z_2 z_1 z_0
(Decimal)	Input		Output
0	0 0	0 0	0 0 0
1	0 0	0 1	0 0 1
2	0 0	1 0	0 1 0
3	0 0	1 1	0 1 1
4	0 1	0 0	0 0 1
5	0 1	0 1	0 1 0
6	0 1	1 0	0 1 1
7	**0 1**	**1 1**	**1 0 0**
8	1 0	0 0	0 1 0
9	1 0	0 1	0 1 1
10	1 0	1 0	1 0 0
11	1 0	1 1	1 0 1
12	1 1	0 0	0 1 1
13	1 1	0 1	1 0 0
14	**1 1**	**1 0**	**1 0 1**
15	1 1	1 1	1 1 0

(b)

Figure 4. Addition table: (a) decimal, (b) binary

Remember that Figure 4*b* describes a 3-bit adder in a tabular format; therefore, in each row Z should be the sum of A and B. For example, consider row 14, which reads: If A = 11 and B = 10, then Z = 101. Let us verify this result: A is binary 11 or decimal 3, and B is binary 10 or decimal 2. So, A + B becomes 3 + 2 = 5 or binary 101, and this is what row 14 reads. Remember that a row number is the decimal equivalent for the corresponding binary input. For example, '14' is the decimal equivalent for the binary

number "1110", which is in the input column of row 14 in Figure 4*b*. Also notice that when you learn binary addition later in this book, you do not have to convert binary "added" and "addend" into decimal, carry out the addition in decimal, and then convert the sum back into binary; you may directly perform additions in binary.

As another example, let us see which row in Figure 4*b* represents 1 + 3, and what the corresponding output is. Here we have A = decimal 1, and B = decimal 3, which correspond to binary 01 and binary 11, respectively. Take a close look at the order of the input variables in Figure 4*b*: a_1 a_0 b_1 b_0. Therefore, the row that we are looking for is identified by the bit pattern 01 11 (01 for a_1 a_0, and 11 for b_1 b_0) in the input column. We use this bit pattern or its decimal equivalent, 7, to index the table to locate the correct row, and eventually determine the right output, which is binary 100 or decimal 4, (1 + 3 = 4). Read this table thoroughly and make sure that you are convinced of the logic used in this table.

To save some room, we may break the table in Figure 4*b* into two halves and then put them side-by-side, as shown in Figure 5.

Row	a_1 a_0	b_1 b_0	z_2 z_1 z_0	Row	a_1 a_0	b_1 b_0	z_2 z_1 z_0
0	0 0	0 0	0 0 0	8	1 0	0 0	0 1 0
1	0 0	0 1	0 0 1	9	1 0	0 1	0 1 1
2	0 0	1 0	0 1 0	10	1 0	1 0	1 0 0
3	0 0	1 1	0 1 1	11	1 0	1 1	1 0 1
4	0 1	0 0	0 0 1	12	1 1	0 0	0 1 1
5	0 1	0 1	0 1 0	13	1 1	0 1	1 0 0
6	0 1	1 0	0 1 1	14	1 1	1 0	1 0 1
7	0 1	1 1	1 0 0	15	1 1	1 1	1 1 0

Figure 5. Side-by-side 2-bit binary addition table

To analyze this table more clearly we may split it into 3 tables, one for each output bit, as shown in Figure 6, Figure 7 and Figure 8 for z_2, z_1 and z_0, respectively. These three 4-variable tables fully describe three 4-variable functions with outputs z_2, z_1 and z_0[3], respectively. (By an n-variable truth table we mean a truth table with n *input* variables.) These tables can be interpreted similar to what we did for Figure 4*b*. For example, row 7 in Figure 8 reads: if a_1 a_0 and b_1 b_0 are 01 and 11, respectively, then the output (z_0) will be 0. Recall from your math courses that, informally speaking, a *function* is basically a manipulation rule that is applied to some input values to produce an output value. An important characteristic of a function is that its output depends only on the *current* value of input; in other words, a function may never generate two different (output) values for the same input value. And this is true for the tables discussed previously; each table (or more specifically, *look-up* table) simply tabulates (in ascending order) all possible input combinations together with the corresponding output bit, so that if we know the input we can directly read the output from the table. Such tables are called *truth tables*. Notice that this tabular format is not a new type of description to us. We are already familiar with, say, the traditional addition table (see Figure 4*a*) and multiplication table, which are based on the same idea.

[3] As a matter of fact, z_0 (LSB of output) is independent of a_1 and b_1 (MSBs of A and B, respectively). This is an obvious fact in decimal additions as well. For example, in "79 + 25 = 104" digit 4 (the least significant digit of the result) is independent of 7 and 2, the most significant digits of the two operands of this addition.

Row	a_1 a_0	b_1 b_0	z_2	Row	a_1 a_0	b_1 b_0	z_2
0	0 0	0 0	0	8	1 0	0 0	0
1	0 0	0 1	0	9	1 0	0 1	0
2	0 0	1 0	0	10	1 0	1 0	1
3	0 0	1 1	0	11	1 0	1 1	1
4	0 1	0 0	0	12	1 1	0 0	0
5	0 1	0 1	0	13	1 1	0 1	1
6	0 1	1 0	0	14	1 1	1 0	1
7	0 1	1 1	1	15	1 1	1 1	1

Figure 6. Truth table of z_2

Row	a_1 a_0	b_1 b_0	z_1	Row	a_1 a_0	b_1 b_0	z_1
0	0 0	0 0	0	8	1 0	0 0	1
1	0 0	0 1	0	9	1 0	0 1	1
2	0 0	1 0	1	10	1 0	1 0	0
3	0 0	1 1	1	11	1 0	1 1	0
4	0 1	0 0	0	12	1 1	0 0	1
5	0 1	0 1	1	13	1 1	0 1	0
6	0 1	1 0	1	14	1 1	1 0	0
7	0 1	1 1	0	15	1 1	1 1	1

Figure 7. Truth table of z_1

Row	a_1 a_0	b_1 b_0	z_0	Row	a_1 a_0	b_1 b_0	z_0
0	0 0	0 0	0	**8**	**1 0**	**0 0**	**0**
1	0 0	0 1	1	9	1 0	0 1	1
2	0 0	1 0	0	10	1 0	1 0	0
3	0 0	1 1	1	11	1 0	1 1	1
4	0 1	0 0	1	12	1 1	0 0	1
5	0 1	0 1	0	13	1 1	0 1	0
6	0 1	1 0	1	14	1 1	1 0	1
7	**0 1**	**1 1**	**0**	15	1 1	1 1	0

Figure 8. Truth table of z_0

Although the same input value cannot generate two or more different outputs, it is quite legitimate to generate identical outputs by applying two or more different inputs. For example, a 0 output is generated in each of rows 5 and 7 in Figure 8, while the inputs are of course different. This is NOT a violation of the definition of a function. ◊

The first step in the manual design of a digital circuit is to convert the problem description (normally in natural language) to a truth table, which can be considered a generic and an unambiguous intermediate language. As soon as a truth table is constructed, the rest of the design procedure can be carried out in a systematic way to be covered in Chapters 3 and 4.

Take a second look at Figure 5. This table represents three truth tables or functions sharing the same input column. In general, input columns of all truth tables with the same number of input variables are identical. More specifically, binary values from 0 to $2^n - 1$ are tabulated in input columns of all n-variable truth tables, resulting in 2^n-row tables. (Remember that there are as many rows in an n-bit truth table as there are n-bit patterns.) The output column, on the other hand, is obviously determined by the function to be described by the truth table. We have already seen 4-variable truth tables with bit patterns 0000 to 1111 in their input columns. The input column of 3-variable truth tables is shown in Figure 9. These tables each have $2^3 = 8$ rows. The table shown in Figure 9 may also be called a 3-variable *blank* truth table, as its output column is blank.

Row	Input	Output
0	000	
1	001	
2	010	
3	011	
4	100	
5	101	
6	110	
7	111	

Figure 9. Three-variable blank truth table

Truth tables grow exponentially with the number of input variables, and this is a major drawback of such tables. More specifically, if you add one more variable, the truth table is doubled in length. As an example, a 3-variable truth table is 8 rows long, while a 4-variable table is 16 rows long. Therefore, not all logic functions may be described by truth tables on paper. In some circumstances, however, it is possible to shrink (long) truth tables using a new notation introduced in Chapter 2. ◊

We should now feel comfortable with different bit patterns, we can conveniently apply all input combinations to our 2-bit adder, and we are confidently able to determine the output of this functional unit for a given input; however, we don't know yet what these 0s and 1s physically mean. In other words, in theory it makes sense to say that at some instant of time the input applied to the adder is 10 11, but what should we really apply to a physically realized adder (with some physical components) in the lab to represent the same bit pattern? Be patient! It will be answered soon.

Example 13. (four-variable *majority function*) Suppose that there is a four-member committee with regular meetings in a four-seat room. Each seat has a sensor that generates a 1 when somebody sits on that seat; a 0 is generated otherwise. Also there is a light outside the room monitoring the situation: it turns on when the majority of the members (i.e., three or more people) attend the meeting (are in their seats), otherwise the light turns off. Further suppose that the light turns on and off with a 1 and a 0, respectively. Obtain a truth table with four input variables (representing the sensors' outputs) and one output (driving the light) to describe a controller for this light.

The truth table of this majority function is shown in Figure 10. Here the four inputs coming from the four sensors are a_3, a_2, a_1 and a_0, and the output is L; i.e., L is the output of a logic function of 4 variables, a_3, a_2, a_1 and a_0. The truth-table construction technique is the same as before. The table has $2^4 = 16$ rows (where 4 is the number of input variables), one input column (with four variables) and one output column. We put all 4-bit patterns (16 in total) in the input column of the table in ascending order. The output column is filled based on our *interpretation* of the problem description provided in natural language. (If

the problem description is not sufficiently clear, then different designers may come up with different interpretations, usually resulting in different truth tables.) We inspect each row individually and assign a 1 to L in any row with 3 or more 1s in the input column; otherwise (if there are 2 or fewer 1s), a 0 is placed in the output column. For example, consider row 5, in which there are only two 1s in the input bit pattern (i.e., only two seats are taken); this row receives a 0 output because this row corresponds to a situation in which only two members attend the meeting. Now consider row 13, in which we have three 1s in the input bit pattern. This represents a situation with over 50% participation, resulting in a 1 in the output column of this row.

Row	a_3 a_2	a_1 a_0	L	Row	a_3 a_2	a_1 a_0	L
0	0 0	0 0	0	8	1 0	0 0	0
1	0 0	0 1	0	9	1 0	0 1	0
2	0 0	1 0	0	10	1 0	1 0	0
3	0 0	1 1	0	11	1 0	1 1	1
4	0 1	0 0	0	12	1 1	0 0	0
5	**0 1**	**0 1**	**0**	**13**	**1 1**	**0 1**	**1**
6	0 1	1 0	0	14	1 1	1 0	1
7	0 1	1 1	1	15	1 1	1 1	1

Figure 10. Truth table of four-bit majority function

Example 14. Consider a chemical plant with two binary output variables, T and P. T is asserted (T = 1) when the temperature rises beyond a threshold, and so is P when the pressure goes above normal. Obtain a truth table for an alarm system with three (input) variables, T, P and O (stands for operator), and one output variable (S), so that when the temperature or the pressure or both go up, the alarm must sound, provided that the plant is not attended by an operator. Notice that the third variable, O, is asserted (O = 1) whenever the plant is supervised by an operator; otherwise O is deasserted. Also assume that the alarm goes off with an asserted output (S = 1).

We employ the same procedure used in Example 13 to develop a truth table. Since there are 3 (input) variables, the truth table would be an eight-row table ($2^3 = 8$) with eight binary numbers, 000 to 111 (in ascending order), in its input column as shown in Figure 11*a*. The output column is filled out based on our understanding from the problem description. To reach this goal, each row is examined individually. For example, consider row 1; neither T nor P is asserted in this row. Therefore, one necessary condition to sound the alarm is not met here; hence the alarm is shut off (S = 0). Additionally, this row lacks the second necessary condition to sound the alarm because the plant is attended in this row. Other rows may be filled out in a similar way to reach the complete table shown in Figure 11*a*.

As more experience is obtained, you will probably develop some shortcuts for filling out truth tables. For this example you may reason as follows:

O = 1: The plant is attended (a sufficient condition to shut off the alarm); so the alarm must not sound, i.e., S = 0. As the first step in filling out the output column we give a 0 to S in any row for which O = 1, i.e., every other row staring with row 1, as shown in Figure 11*b*.

Among the remaining rows, if neither the temperature nor the pressure is high, i.e., T = 0 and P = 0 (which only corresponds to row 0), then the alarm must not sound (see Figure 11*c*). Otherwise, if at least one of these variables is asserted, then S has to be asserted as well, as shown in Figure 11*d*. ◊

Row	T	P	O	S
0	0	0	0	0
1	**0**	**0**	**1**	**0**
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	0

(a)

Row	T	P	O	S
0	0	0	0	
1	0	0	1	0
2	0	1	0	
3	0	1	1	0
4	1	0	0	
5	1	0	1	0
6	1	1	0	
7	1	1	1	0

(b)

Row	T	P	O	S
0	0	0	0	0
1	0	0	1	0
2	0	1	0	
3	0	1	1	0
4	1	0	0	
5	1	0	1	0
6	1	1	0	
7	1	1	1	0

(c)

Row	T	P	O	S
0	0	0	0	0
1	0	0	1	0
2	0	1	0	
3	0	1	1	0
4	1	0	0	
5	1	0	1	0
6	1	1	0	
7	1	1	1	0

(d)

Figure 11. Chemical plant example: (a) complete truth table, (b) attended plant, (c) unattended plant with normal T and P, (d) unattended plant with high temperature or high pressure or both

The following are some true statements regarding this truth table. Similar statements are commonly used in logic analysis and design.

- O = 1 is *sufficient* to shut off the alarm (S = 0). This means that if O = 1, no matter what other input bits are, the alarm must not sound.
- O = 0 is *necessary* to turn the alarm on. This means that an on alarm is necessarily unattended.
- One asserted T or P is *necessary* to sound the alarm, but it is *not sufficient*.
- While O = 0, it is *necessary* and *sufficient* to have one asserted T or P to sound the alarm.

Physical Meanings of logic 1 and logic 0

Let us point out and then answer two major questions that should have arisen:

Question: How can we physically construct a digital circuit, say, a 2-bit adder mentioned previously, to satisfy its terminal behavior described by its truth table(s)?

Answer: *Gates* or the basic building blocks of digital circuits will be introduced in Chapter 2. Then a design methodology will be developed in Chapters 3 and 4. In the manual design (as opposed to computerized design) we use this design methodology and also a menu of gates. The methodology helps us properly pick what we need from the menu and put them together to reach our goal, say, a 2-bit adder, once we have the truth table(s) ready. Only the manual design of digital circuits is covered in this book.

Question: What are the *physical* meanings of logic 1 and logic 0 in digital systems? In other words, what should we do with an input line, *a*, so that a physical circuit recognizes it as a logic 1 or a logic 0?

Answer: In today's technologies logic 1 and 0 are mostly mapped to two different *voltage* bands, namely V_{high} or H for short, and V_{low} or L for short. For example, in Transistor-Transistor Logic (TTL) technology the voltage bands are approximately as follows:

2 volts < H < 5 volts and 0 volt < L < 0.8 volts

These voltage bands are also shown in Figure 12.

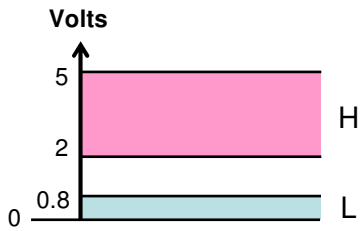

Figure 12. Approximate voltage bands for TTL

There are obviously two possible mappings between 0, 1 and H, L. If 0 is assigned to L and 1 is assigned to H, then the variables are called *high-asserting*. Variables with the opposite assignment are called *low-asserting* (i.e., when 1 is assigned to L and 0 is assigned to H). In *positive* logic all variables are high-asserting, while in *negative* logic all variables are low-asserting. Therefore, in a positive-logic truth table if 0 and 1 are swapped throughout (i.e., all 1s are replaced with 0s and all 0s are replaced with 1s), a negative-logic truth table will be obtained. The same replacement will convert a negative-logic truth table to a positive-logic one. Figure 13*b* shows a negative-logic truth table for the positive-logic one shown in Figure 11*a*, which has been repeated in Figure 13*a* for ease of reference. The rows of the table in Figure 13*b* have been rearranged in ascending order in Figure 13*c*. Remember that a row number is always the decimal equivalent for the corresponding binary number in the input column.

Row	T	P	O	S
0	0	0	0	0
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	0

(a)

Row	T	P	O	S
7	1	1	1	1
6	1	1	0	1
5	1	0	1	0
4	1	0	0	1
3	0	1	1	0
2	0	1	0	1
1	0	0	1	0
0	0	0	0	1

(b)

Row	T	P	O	S
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

(c)

T	P	O	S
L	L	L	L
L	L	H	L
L	H	L	H
L	H	H	L
H	L	L	H
H	L	H	L
H	H	L	H
H	H	H	L

(d)

Figure 13. (a) Positive-logic truth table, (b) , (c) negative-logic truth table, (d) voltage table

By introducing H and L as the electrical counterparts of logic values, an electrical counterpart called *voltage table* is created for truth tables. Remember that a voltage table is independent of the assertion levels of the participating variables and remains part of the characteristics of the underlying logic circuit. The voltage table is obtained by simply replacing 1s with Hs and 0s with Ls in the positive-logic truth table. Similarly, it can be obtained by replacing 0s with Hs and 1s with Ls in the negative-logic truth table. Figure 13*d* illustrates a voltage table for the truth tables shown in Figure 13*a*, Figure 13*b* and Figure 13*c*.

High-asserting and low-asserting variables may be used together in the same circuit, expression and/or truth table resulting in *mixed* logic. In this book we use positive logic unless otherwise specified.

The positive-logic truth table shown in Figure 11*a* has been redrawn in Figure 14 as a mixed-logic truth table in which variables P and S are low-asserting and variables T and O are high-asserting. Remember that to convert a high-asserting variable (in a truth table) to a low-asserting one or vice versa, all 1s in the truth table are replaced with 0s and all 0s in the truth table are replaced with 1s for that specific variable.

Row	T	P	O	S
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

Figure 14. Mixed-logic truth table corresponding to Figure 13 (P and S are low-asserting)