chapter

3

Number Representation and Arithmetic Circuits

Chapter Objectives

In this chapter you will learn about:

- Representation of numbers in computers
- Circuits used to perform arithmetic operations
- Performance issues in large circuits
- Use of Verilog to specify arithmetic circuits

In this chapter we will discuss logic circuits that perform arithmetic operations. We will explain how numbers can be added, subtracted, and multiplied. We will also show how to write Verilog code to describe the arithmetic circuits. These circuits provide an excellent platform for illustrating the power and versatility of Verilog in specifying complex logic-circuit assemblies. The concepts involved in the design of arithmetic circuits are easily applied to a wide variety of other circuits.

Before tackling the design of arithmetic circuits, it is necessary to discuss how numbers are represented in digital systems. In Chapter 1 we introduced binary numbers and showed how they can be expressed using the positional number representation. We also discussed the conversion process between decimal and binary number systems. In Chapter 2 we dealt with logic variables in a general way, using variables to represent either the states of switches or some general conditions. Now we will use the variables to represent numbers. Several variables are needed to specify a number, with each variable corresponding to one digit of the number.

3.1 POSITIONAL NUMBER REPRESENTATION

|导读|

本节介绍了标准的正整数(即无符号整数)的数位表示法,然后阐述了八进制数和十六进制数的表示方法。

When dealing with numbers and arithmetic operations, it is convenient to use standard symbols. Thus to represent addition we use the plus (+) symbol, and for subtraction we use the minus (−) symbol. In Chapter 2 we used the + symbol mostly to represent the logical OR operation. Even though we will now use the same symbols for two different purposes, the meaning of each symbol will usually be clear from the context of the discussion. In cases where there may be some ambiguity, the meaning will be stated explicitly.

3.1.1 UNSIGNED INTEGERS

The simplest numbers to consider are the integers. We will begin by considering positive integers and then expand the discussion to include negative integers. Numbers that are positive only are called *unsigned*, and numbers that can also be negative are called *signed*. Representation of numbers that include a radix point (real numbers) is discussed later in the chapter.

As explained in Section 1.5.1, an n-bit unsigned number

$$B = b_{n-1}b_{n-2}\cdots b_1b_0$$

represents an integer that has the value

无符号数:

Unsigned number,只有正值的数。

$$V(B) = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0 \qquad [3.1]$$
$$= \sum_{i=0}^{n-1} b_i \times 2^i$$

3.1.2 OCTAL AND HEXADECIMAL REPRESENTATIONS

The positional number representation can be used for any radix. If the radix is r, then the number

$$K = k_{n-1}k_{n-2}\cdots k_1k_0$$

has the value

$$V(K) = \sum_{i=0}^{n-1} k_i \times r^i$$

有符号数：
Signed number,可以有负值的数。

Our interest is limited to those radices that are most practical. We will use decimal numbers because they are used by people, and we will use binary numbers because they are used by computers. In addition, two other radices are useful—8 and 16. Numbers represented with radix 8 are called *octal* numbers, while radix-16 numbers are called *hexadecimal* numbers. In octal representation the digit values range from 0 to 7. In hexadecimal representation (often abbreviated as *hex*), each digit can have one of 16 values. The first ten are denoted the same as in the decimal system, namely, 0 to 9. Digits that correspond to the decimal values 10, 11, 12, 13, 14, and 15 are denoted by the letters, A, B, C, D, E, and F. Table 3.1 gives the first 18 integers in these number systems.

In computers the dominant number system is binary. The reason for using the octal and hexadecimal systems is that they serve as a useful shorthand notation for binary numbers. One octal digit represents three bits. Thus a binary number is converted into an octal number

Table 3.1 Numbers in different systems.

Decimal	Binary	Octal	Hexadecimal
00	00000	00	00
01	00001	01	01
02	00010	02	02
03	00011	03	03
04	00100	04	04
05	00101	05	05
06	00110	06	06
07	00111	07	07
08	01000	10	08
09	01001	11	09
10	01010	12	0A
11	01011	13	0B
12	01100	14	0C
13	01101	15	0D
14	01110	16	0E
15	01111	17	0F
16	10000	20	10
17	10001	21	11
18	10010	22	12

数位表示法：
Positional Number Representation,用数字所在的位置及其权重来表示一个数的方法。

by taking groups of three bits, starting from the least-significant bit, and replacing them with the corresponding octal digit. For example, 101011010111 is converted as

$$\underbrace{101}_{5} \quad \underbrace{011}_{3} \quad \underbrace{010}_{2} \quad \underbrace{111}_{7}$$

which means that $(101011010111)_2 = (5327)_8$. If the number of bits is not a multiple of three, then we add 0s to the left of the most-significant bit. For example, $(10111011)_2 = (273)_8$ because of the grouping

八进制数：
Octal number,基为8的数。

$$\underbrace{010}_{2} \quad \underbrace{111}_{7} \quad \underbrace{011}_{3}$$

Conversion from octal to binary is just as straightforward; each octal digit is simply replaced by three bits that denote the same value.

Similarly, a hexadecimal digit represents four bits. For example, a 16-bit number is represented by four hex digits, as in

$$(1010111100100101)_2 = (AF25)_{16}$$

using the grouping

$$\underbrace{1010}_{A} \quad \underbrace{1111}_{F} \quad \underbrace{0010}_{2} \quad \underbrace{0101}_{5}$$

Zeros are added to the left of the most-significant bit if the number of bits is not a multiple of four. For example, $(1101101000)_2 = (368)_{16}$ because of the grouping

$$\underbrace{0011}_{3} \quad \underbrace{0110}_{6} \quad \underbrace{1000}_{8}$$

十六进制数：
Hexadecimal number,基为16的数。

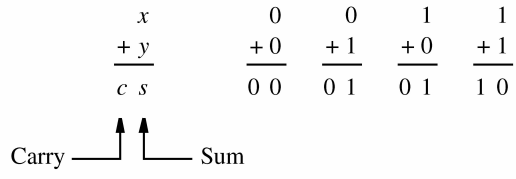
Conversion from hexadecimal to binary involves straightforward substitution of each hex digit by four bits that denote the same value.

Binary numbers used in modern computers often have 32 or 64 bits. Written as binary n-tuples (sometimes called bit vectors), such numbers are awkward for people to deal with. It is much simpler to deal with them in the form of 8- or 16-digit hex numbers. Because the arithmetic operations in a digital system usually involve binary numbers, we will focus on circuits that use such numbers. We will sometimes use the hexadecimal representation as a convenient shorthand description.

We have introduced the simplest numbers—unsigned integers. It is necessary to be able to deal with several other types of numbers. We will discuss the representation of signed numbers, fixed-point numbers, and floating-point numbers later in this chapter. But first we will examine some simple circuits that operate on numbers to give the reader a feeling for digital circuits that perform arithmetic operations and to provide motivation for further discussion.

3.2 ADDITION OF UNSIGNED NUMBERS

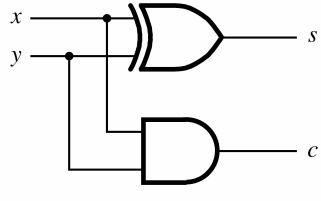
Binary addition is performed in the same way as decimal addition except that the values of individual digits can be only 0 or 1. In Chapter 2, we already considered the addition of 2 one-bit numbers, as an example of a simple logic circuit. Now, we will consider this task in the context of general adder circuits. The one-bit addition entails four possible combinations, as indicated in Figure 3.1*a*. Two bits are needed to represent the result of the addition. The right-most bit is called the *sum*, *s*. The left-most bit, which is produced as a carry-out when both bits being added are equal to 1, is called the *carry*, *c*. The addition operation is defined in the form of a truth table in part (*b*) of the figure. The sum bit *s* is the XOR function. The carry *c* is the AND function of inputs *x* and *y*. A circuit realization of these functions is shown in Figure 3.1*c*. This circuit, which implements the addition of only two bits, is called a *half-adder*.



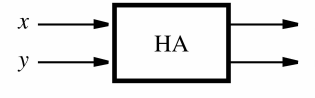
(a) The four possible cases

x	y	Carry c	Sum s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(b) Truth table



(c) Circuit



(d) Graphical symbol

Figure 3.1 Half-adder.

|导读|

本节首先讲述了无符号数的加法运算及其对应的半加器和全加器电路，并用异或门来实现加法运算的和输出，以及异或和同或的特殊性质。然后将全加器分解成两个半加器组成的电路，并将多个全加器级联，进位信号像波浪一样在其中传播，构成了行波进位加法器。最后，通过一个设计实例巧妙地说明了用移位相加的运算方式来完成一个8位无符号数乘以3的乘法运算。

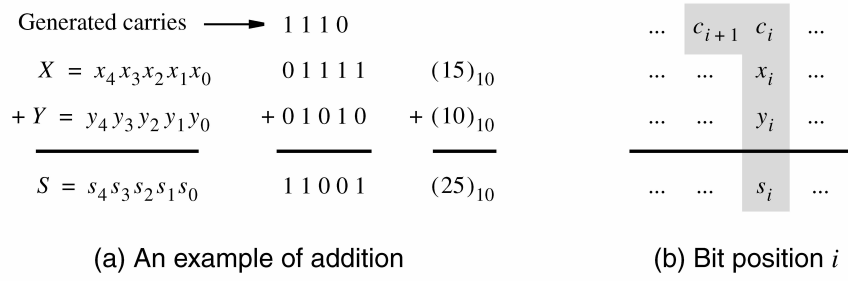


Figure 3.2 Addition of multibit numbers.

A more interesting case is when larger numbers that have multiple bits are involved. Then it is still necessary to add each pair of bits, but for each bit position i, the addition operation may include a *carry-in* from bit position $i - 1$.

Figure 3.2*a* presents an example of the addition operation. The two operands are $X = (01111)_2 = (15)_{10}$ and $Y = (01010)_2 = (10)_{10}$. Five bits are used to represent X and Y, making it possible to represent integers in the range from 0 to 31; hence the sum $S = X + Y = (25)_{10}$ can also be denoted as a five-bit integer. Note the labeling of individual bits, such that $X = x_4x_3x_2x_1x_0$ and $Y = y_4y_3y_2y_1y_0$. The figure shows, in a blue color, the carries generated during the addition process. For example, a carry of 0 is generated when x_0 and y_0 are added, a carry of 1 is produced when x_1 and y_1 are added, and so on.

半加器：
Half adder(HA),只对两个1比特数进行加运算的电路。

In Chapter 2 we designed logic circuits by first specifying their behavior in the form of a truth table. This approach is impractical in designing an adder circuit that can add the five-bit numbers in Figure 3.2. The required truth table would have 10 input variables, 5 for each number X and Y. It would have $2^{10} = 1024$ rows! A better approach is to consider the addition of each pair of bits, x_i and y_i, separately.

For bit position 0, there is no carry-in, and hence the addition is the same as for Figure 3.1. For each other bit position i, the addition involves bits x_i and y_i, and a carry-in c_i, as illustrated in Figure 3.2*b*. This observation leads to the design of a logic circuit that has three inputs x_i, y_i, and c_i, and produces the two outputs s_i and c_{i+1}. The required truth table is shown in Figure 3.3*a*. The sum bit, s_i, is the modulo-2 sum of x_i, y_i, and c_i. The *carry-out*, c_{i+1}, is equal to 1 if the sum of x_i, y_i, and c_i is equal to either 2 or 3. Karnaugh maps for these functions are shown in part (*b*) of the figure. For the carry-out function the optimal sum-of-products realization is

$$c_{i+1} = x_iy_i + x_ic_i + y_ic_i$$

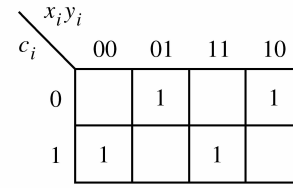
For the s_i function a sum-of-products realization is

$$s_i = \overline{x}_iy_i\overline{c}_i + x_i\overline{y}_i\overline{c}_i + \overline{x}_i\overline{y}_ic_i + x_iy_ic_i$$

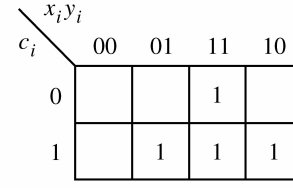
A more attractive way of implementing this function is by using the XOR gates, as explained below.

c_i	x_i	y_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a) Truth table

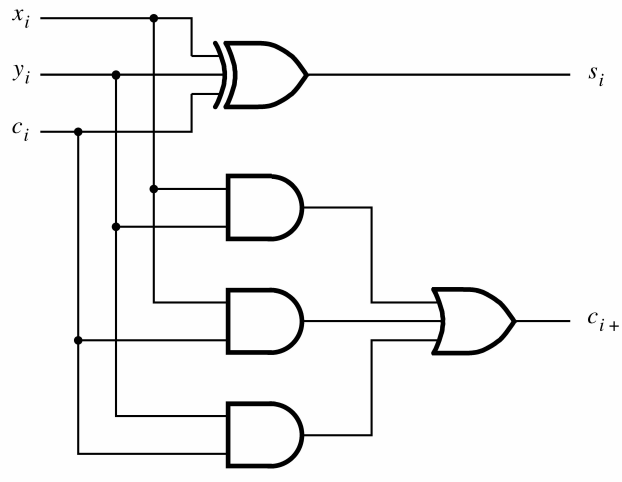


$$s_i = x_i \oplus y_i \oplus c_i$$



$$c_{i+1} = x_iy_i + x_ic_i + y_ic_i$$

(b) Karnaugh maps



(c) Circuit

Figure 3.3 Full-adder.

全加器：

Full adder(FA),只对两个多比特数的某一位进行加运算的电路，要考虑低位给本位的进位，以及本位向高位的进位。

Use of XOR Gates

As shown in Chapter 2, the XOR function of two variables is defined as $x_1 \oplus x_2 = \overline{x}_1 x_2 + x_1 \overline{x}_2$. The preceding expression for the sum bit can be manipulated into a form that uses only XOR operations as follows

$$\begin{aligned} s_i &= (\overline{x}_i y_i + x_i \overline{y}_i)\overline{c}_i + (\overline{x}_i \overline{y}_i + x_i y_i)c_i \\ &= (x_i \oplus y_i)\overline{c}_i + \overline{(x_i \oplus y_i)}c_i \\ &= (x_i \oplus y_i) \oplus c_i \end{aligned}$$

The XOR operation is associative; hence we can write

$$s_i = x_i \oplus y_i \oplus c_i$$

Therefore, a three-input XOR operation can be used to realize s_i.

The XOR operation generates as an output a modulo-2 sum of its inputs. Thus, the output is equal to 1 if an odd number of inputs have the value 1, and it is equal to 0 otherwise. For this reason the XOR is sometimes referred to as the *odd* function. Observe that the XOR has no minterms that can be combined into a larger product term, as evident from the checkerboard pattern for function s_i in the map in Figure 3.3*b*. The logic circuit implementing the truth table in Figure 3.3*a* is given in Figure 3.3*c*. This circuit is known as a *full-adder*.

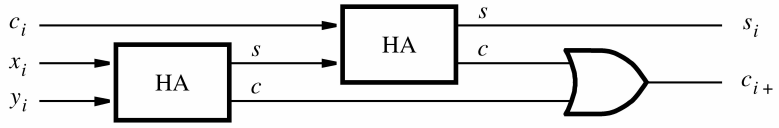
奇函数：
Odd function,即异或运算,输入中1的个数为奇数时,输出为1。

Another interesting feature of XOR gates is that a two-input XOR gate can be thought of as using one input as a control signal that determines whether the true or complemented value of the other input will be passed through the gate as the output value. This is clear from the definition of XOR, where $x_i \oplus y_i = \overline{x}y + x\overline{y}$. Consider x to be the control input. Then if $x = 0$, the output will be equal to the value of y. But if $x = 1$, the output will be equal to the complement of y. In the derivation above, we used algebraic manipulation to derive $s_i = (x_i \oplus y_i) \oplus c_i$. We could have obtained the same expression immediately by making the following observation. In the top half of the truth table in Figure 3.3*a*, c_i is equal to 0, and the sum function s_i is the XOR of x_i and y_i. In the bottom half of the table, c_i is equal to 1, while s_i is the complemented version of its top half. This observation leads directly to our expression using 2 two-input XOR operations. We will encounter an important example of using XOR gates to pass true or complemented signals under the control of another signal in Section 3.3.3.

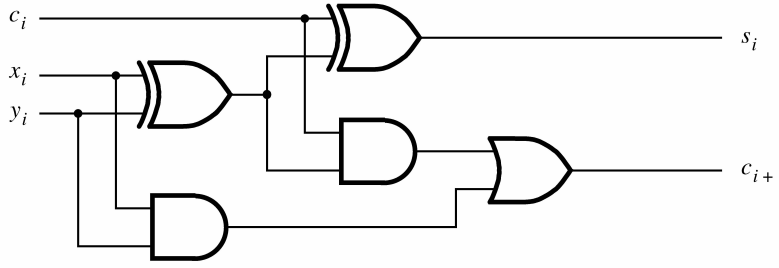
In the preceding discussion we encountered the complement of the XOR operation, which we denoted as $\overline{x \oplus y}$. This operation is used so commonly that it is given the distinct name *XNOR*. A special symbol, $\odot$, is often used to denote the XNOR operation, namely

$$x \odot y = \overline{x \oplus y}$$

The XNOR is sometimes also referred to as the *coincidence* operation because it produces the output of 1 when its inputs coincide in value; that is, they are both 0 or both 1.



(a) Block diagram



(b) Detailed diagram

Figure 3.4 A decomposed implementation of the full-adder circuit.

一致运算：
Coincidence operation，即同或运算，当两个输入的取值一致（同为0或者1）时，输出为1。

3.2.1 DECOMPOSED FULL-ADDER

In view of the names used for the circuits, one can expect that a full-adder can be constructed using half-adders. This can be accomplished by creating a multilevel circuit given in Figure 3.4. It uses two half-adders to form a full-adder. The reader should verify the functional correctness of this circuit.

3.2.2 RIPPLE-CARRY ADDER

To perform addition by hand, we start from the least-significant digit and add pairs of digits, progressing to the most-significant digit. If a carry is produced in position i, then this carry is added to the operands in position $i + 1$. The same arrangement can be used in a logic circuit that performs addition. For each bit position we can use a full-adder circuit, connected as shown in Figure 3.5. Note that to be consistent with the customary way of writing numbers, the least-significant bit position is on the right. Carries that are produced by the full-adders propagate to the left.

When the operands X and Y are applied as inputs to the adder, it takes some time before the output sum, S, is valid. Each full-adder introduces a certain delay before its s_i and c_{i+1} outputs are valid. Let this delay be denoted as Δt. Thus the carry-out from the first stage, c_1, arrives at the second stage Δt after the application of the x_0 and y_0 inputs. The carry-out from the second stage, c_2, arrives at the third stage with a $2\Delta t$ delay, and so on. The signal c_{n-1} is valid after a delay of $(n - 1)\Delta t$, which means that the complete sum is available

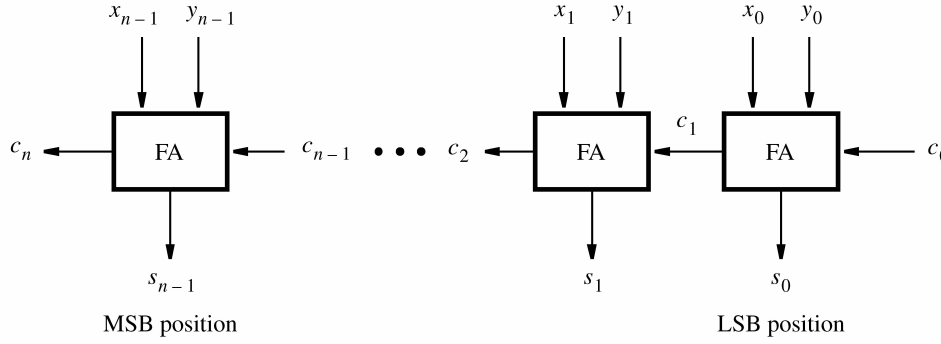


Figure 3.5 An n-bit ripple-carry adder.

after a delay of $n\Delta t$. Because of the way the carry signals "ripple" through the full-adder stages, the circuit in Figure 3.5 is called a *ripple-carry adder*.

行波进位加法器：Ripple carry adder，多个全加器级联，低位向高位的进位接入高位全加器的来自低位的进位端，进位信号像波浪一样传播。

The delay incurred to produce the final sum and carry-out in a ripple-carry adder depends on the size of the numbers. When 32- or 64-bit numbers are used, this delay may become unacceptably high. Because the circuit in each full-adder leaves little room for a drastic reduction in the delay, it may be necessary to seek different structures for implementation of n-bit adders. We will discuss a technique for building high-speed adders in Section 3.4.

So far we have dealt with unsigned integers only. The addition of such numbers does not require a carry-in for stage 0. In Figure 3.5 we included c_0 in the diagram so that the ripple-carry adder can also be used for subtraction of numbers, as we will see in Section 3.3.

3.2.3 DESIGN EXAMPLE

Suppose that we need a circuit that multiplies an eight-bit unsigned number by 3. Let $A = a_7a_6 \cdots a_1a_0$ denote the number and $P = p_9p_8 \cdots p_1p_0$ denote the product $P = 3A$. Note that 10 bits are needed to represent the product.

A simple approach to design the required circuit is to use two ripple-carry adders to add three copies of the number A, as illustrated in Figure 3.6*a*. The symbol that denotes each adder is a commonly-used graphical symbol for adders. The letters x_i, y_i, s_i, and c_i indicate the meaning of the inputs and outputs according to Figure 3.5. The first adder produces $A + A = 2A$. Its result is represented as eight sum bits and the carry from the most-significant bit. The second adder produces $2A + A = 3A$. It has to be a nine-bit adder to be able to handle the nine bits of 2A, which are generated by the first adder. Because the y_i inputs have to be driven only by the eight bits of A, the ninth input y_8 is connected to a constant 0.

This approach is straightforward, but not very efficient. Because $3A = 2A + A$, we can observe that $2A$ can be generated by shifting the bits of A one bit-position to the left, which gives the bit pattern $a_7a_6a_5a_4a_3a_2a_1a_0$0. According to Equation 3.1, this pattern is