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

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

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

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

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

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

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

$$\underbrace{1\,0\,1}_{5} \qquad \underbrace{0\,1\,1}_{3} \qquad \underbrace{0\,1\,0}_{2} \qquad \underbrace{1\,1\,1}_{7}$$

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

$$\underbrace{0\,1\,0}_{2} \qquad \underbrace{1\,1\,1}_{7} \qquad \underbrace{0\,1\,1}_{3}$$

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

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

using the grouping

$$\underbrace{1\,0\,1\,0}_{\mathrm{A}} \qquad \underbrace{1\,1\,1\,1}_{\mathrm{F}} \qquad \underbrace{0\,0\,1\,0}_{2} \qquad \underbrace{0\,1\,0\,1}_{5}$$

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

$$\underbrace{0\,0\,1\,1}_{3} \qquad \underbrace{0\,1\,1\,0}_{6} \qquad \underbrace{1\,0\,0\,0}_{8}$$

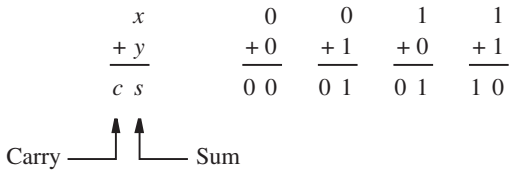
Conversion from hexadecimal to binary involves straightforward substitution of each hex digit by four bits that denote the same value.

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

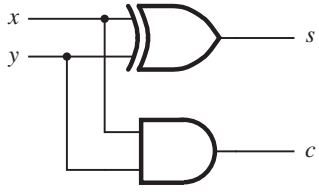
Binary addition is performed in the same way as decimal addition except that the values of individual digits can be only 0 or 1. In Chapter 2, we already considered the addition of 2 one-bit numbers, as an example of a simple logic circuit. Now, we will consider this task in the context of general adder 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

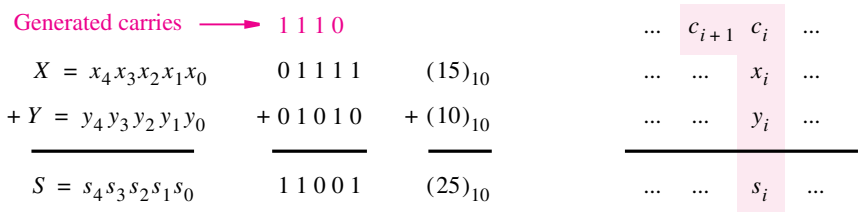
(b) Truth table



(c) Circuit

(d) Graphical symbol

Figure 3.1 Half-adder.



(a) An example of addition (b) Bit position i

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.

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

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

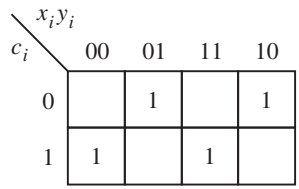
For the s_i function a sum-of-products realization is

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

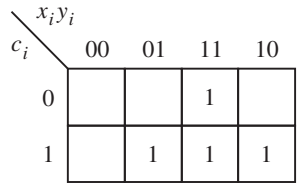
A more attractive way of implementing this function is by using the XOR gates, as explained below.

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

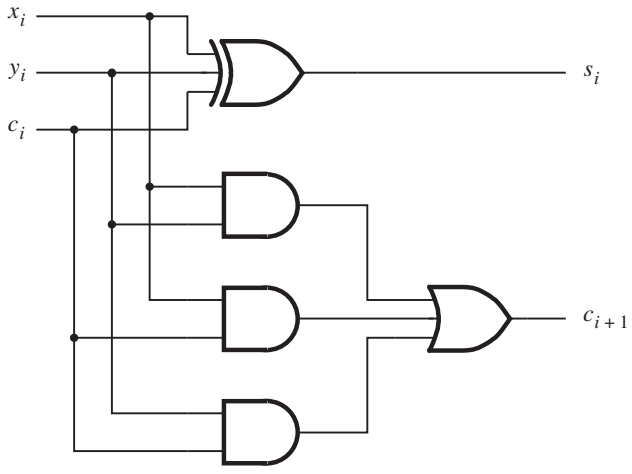


$$s_i = x_i \oplus y_i \oplus c_i$$



$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

(b) Karnaugh maps



(c) Circuit

Figure 3.3 Full-adder.

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*.

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.

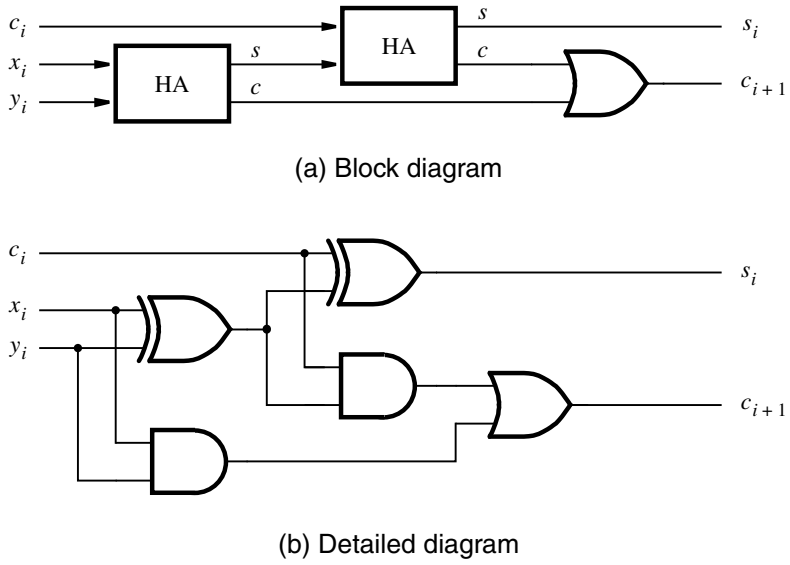


Figure 3.4 A decomposed implementation of the full-adder circuit.

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

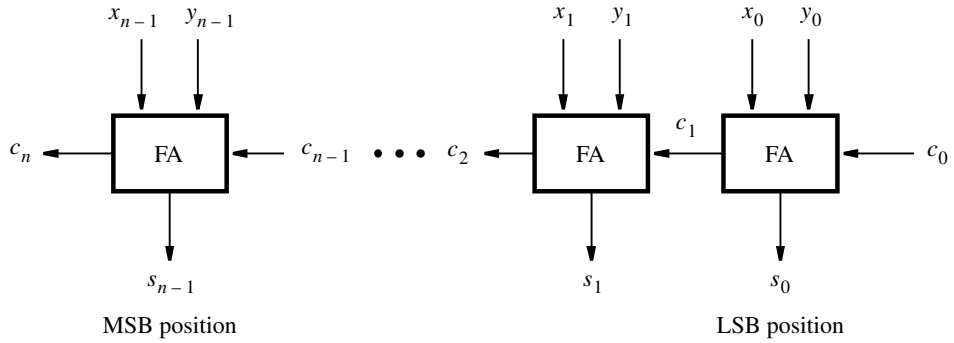


Figure 3.5 An n-bit ripple-carry adder.

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*.

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.

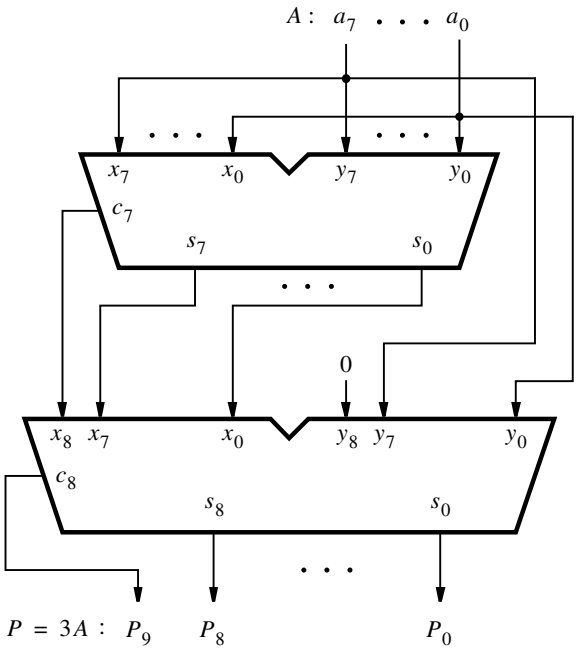
So far we have dealt with unsigned integers only. The addition of such numbers does not require a carry-in for stage 0. In Figure 3.5 we included c_0 in the diagram so that the ripple-carry adder can also be used for subtraction of numbers, as we will see in Section 3.3.

3.2.3 DESIGN EXAMPLE

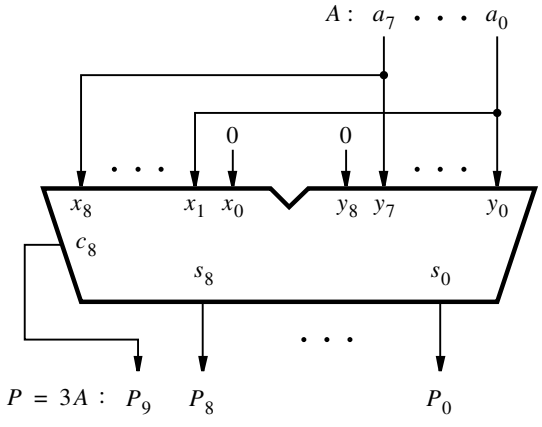
Suppose that we need a circuit that multiplies an eight-bit unsigned number by 3. Let $A = a_7a_6 \cdots a_1a_0$ denote the number and $P = p_9p_8 \cdots p_1p_0$ denote the product $P = 3A$. Note that 10 bits are needed to represent the product.

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 3.6a. 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



(a) Naive approach



(b) Efficient design

Figure 3.6 Circuit that multiplies an eight-bit unsigned number by 3.

equal to 2*A*. Then a single ripple-carry adder suffices for implementing 3*A*, as shown in Figure 3.6*b*. This is essentially the same circuit as the second adder in part (*a*) of the figure. Note that the input x_0 is connected to a constant 0. Note also that in the second adder in part (*a*) of the figure the value of x_0 is always 0, even though it is driven by the least-significant bit, s_0, of the sum of the first adder. Because $x_0 = y_0 = a_0$ in the first adder, the sum bit s_0 will be 0, whether a_0 is 0 or 1.

3.3 SIGNED NUMBERS

In the decimal system the sign of a number is indicated by a + or − symbol to the left of the most-significant digit. In the binary system the *sign* of a number is denoted by the left-most bit. For a positive number the left-most bit is equal to 0, and for a negative number it is equal to 1. Therefore, in signed numbers the left-most bit represents the sign, and the remaining $n - 1$ bits represent the magnitude, as illustrated in Figure 3.7. It is important to note the difference in the location of the most-significant bit (MSB). In unsigned numbers all bits represent the magnitude of a number; hence all n bits are *significant* in defining the magnitude. Therefore, the MSB is the left-most bit, b_{n-1}. In signed numbers there are $n - 1$ significant bits, and the MSB is in bit position b_{n-2}.

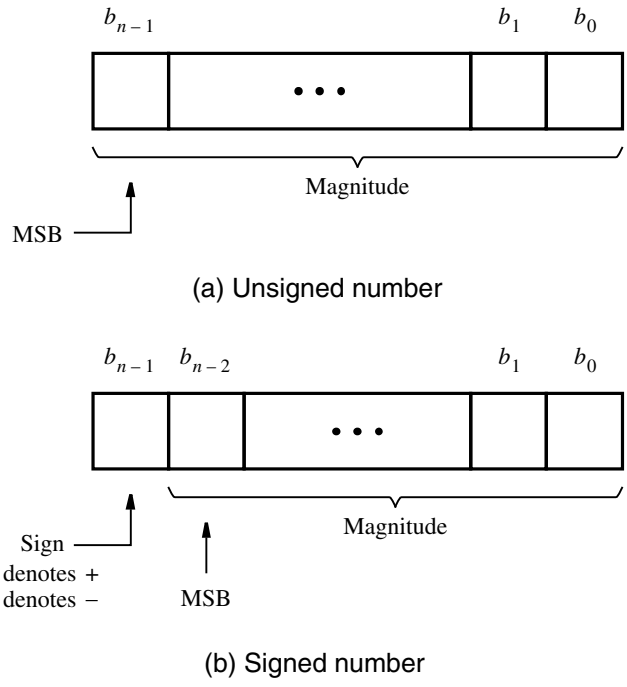


Figure 3.7 Formats for representation of integers.

3.3.1 NEGATIVE NUMBERS

Positive numbers are represented using the positional number representation as explained in the previous section. Negative numbers can be represented in three different ways: sign-and-magnitude, 1's complement, and 2's complement.

Sign-and-Magnitude Representation

In the familiar decimal representation, the magnitude of both positive and negative numbers is expressed in the same way. The sign symbol distinguishes a number as being positive or negative. This scheme is called the *sign-and-magnitude* number representation. The same scheme can be used with binary numbers in which case the sign bit is 0 or 1 for positive or negative numbers, respectively. For example, if we use four-bit numbers, then $+5 = 0101$ and $-5 = 1101$. Because of its similarity to decimal sign-and-magnitude numbers, this representation is easy to understand. However, as we will see shortly, this representation is not well suited for use in computers. More suitable representations are based on complementary systems, explained below.

1's Complement Representation

In a complementary number system, the negative numbers are defined according to a subtraction operation involving positive numbers. We will consider two schemes for binary numbers: the 1's complement and the 2's complement. In the *1's complement* scheme, an n-bit negative number, K, is obtained by subtracting its equivalent positive number, P, from $2^n - 1$; that is, $K = (2^n - 1) - P$. For example, if $n = 4$, then $K = (2^4 - 1) - P = (15)_{10} - P = (1111)_2 - P$. If we convert $+5$ to a negative, we get $-5 = 1111 - 0101 = 1010$. Similarly, $+3 = 0011$ and $-3 = 1111 - 0011 = 1100$. Clearly, the 1's complement can be obtained simply by complementing each bit of the number, including the sign bit. While 1's complement numbers are easy to derive, they have some drawbacks when used in arithmetic operations, as we will see in the next section.

2's Complement Representation

In the 2's complement scheme, a negative number, K, is obtained by subtracting its equivalent positive number, P, from 2^n; namely, $K = 2^n - P$. Using our four-bit example, $-5 = 10000 - 0101 = 1011$, and $-3 = 10000 - 0011 = 1101$. Finding 2's complements in this manner requires performing a subtraction operation that involves borrows. However, we can observe that if K_1 is the 1's complement of P and K_2 is the 2's complement of P, then

$$K_1 = (2^n - 1) - P$$
$$K_2 = 2^n - P$$

It follows that $K_2 = K_1 + 1$. Thus a simpler way of finding a 2's complement of a number is to add 1 to its 1's complement because finding a 1's complement is trivial. This is how 2's complement numbers are obtained in logic circuits that perform arithmetic operations.

The reader will need to develop an ability to find 2's complement numbers quickly. There is a simple rule that can be used for this purpose.

Rule for Finding 2's Complements

Given a number $B = b_{n-1}b_{n-2}\cdots b_1b_0$, its 2's complement, $K = k_{n-1}k_{n-2}\cdots k_1k_0$, can be found by examining the bits of B from right to left and taking the following action: copy all bits of B that are 0 and the first bit that is 1; then simply complement the rest of the bits.

For example, if $B = 0110$, then we copy $k_0 = b_0 = 0$ and $k_1 = b_1 = 1$, and complement the rest so that $k_2 = \overline{b}_2 = 0$ and $k_3 = \overline{b}_3 = 1$. Hence $K = 1010$. As another example, if $B = 10110100$, then $K = 01001100$. We leave the proof of this rule as an exercise for the reader.

Table 3.2 illustrates the interpretation of all 16 four-bit patterns in the three signed-number representations that we have considered. Note that for both sign-and-magnitude representation and for 1's complement representation there are two patterns that represent the value zero. For 2's complement there is only one such pattern. Also, observe that the range of numbers that can be represented with four bits in 2's complement form is -8 to $+7$, while in the other two representations it is -7 to $+7$.

Using 2's-complement representation, an n-bit number $B = b_{n-1}b_{n-2}\cdots b_1b_0$ represents the value

$$V(B) = (-b_{n-1} \times 2^{n-1}) + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0 \qquad \textbf{[3.2]}$$

Thus the largest negative number, $100\ldots00$, has the value -2^{n-1}. The largest positive number, $011\ldots11$, has the value $2^{n-1} - 1$.

Table 3.2 Interpretation of four-bit signed integers.

$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0111	+7	+7	+7
0110	+6	+6	+6
0101	+5	+5	+5
0100	+4	+4	+4
0011	+3	+3	+3
0010	+2	+2	+2
0001	+1	+1	+1
0000	+0	+0	+0
1000	−0	−7	−8
1001	−1	−6	−7
1010	−2	−5	−6
1011	−3	−4	−5
1100	−4	−3	−4
1101	−5	−2	−3
1110	−6	−1	−2
1111	−7	−0	−1

3.3.2 ADDITION AND SUBTRACTION

To assess the suitability of different number representations, it is necessary to investigate their use in arithmetic operations—particularly in addition and subtraction. We can illustrate the good and bad aspects of each representation by considering very small numbers. We will use four-bit numbers, consisting of a sign bit and three significant bits. Thus the numbers have to be small enough so that the magnitude of their sum can be expressed in three bits, which means that the sum cannot exceed the value 7.

Addition of positive numbers is the same for all three number representations. It is actually the same as the addition of unsigned numbers discussed in Section 3.2. But there are significant differences when negative numbers are involved. The difficulties that arise become apparent if we consider operands with different combinations of signs.

Sign-and-Magnitude Addition

If both operands have the same sign, then the addition of sign-and-magnitude numbers is simple. The magnitudes are added, and the resulting sum is given the sign of the operands. However, if the operands have opposite signs, the task becomes more complicated. Then it is necessary to subtract the smaller number from the larger one. This means that logic circuits that compare and subtract numbers are also needed. We will see shortly that it is possible to perform subtraction without the need for this circuitry. For this reason, the sign-and-magnitude representation is not used in computers.

1's Complement Addition

An obvious advantage of the 1's complement representation is that a negative number is generated simply by complementing all bits of the corresponding positive number. Figure 3.8 shows what happens when two numbers are added. There are four cases to consider in terms of different combinations of signs. As seen in the top half of the figure, the computation of $5 + 2 = 7$ and $(-5) + 2 = (-3)$ is straightforward; a simple addition of the operands gives the correct result. Such is not the case with the other two possibilities. Computing $5 + (-2) = 3$ produces the bit vector 10010. Because we are dealing with four-bit numbers, there is a carry-out from the sign-bit position. Also, the four bits of the

(+5)	0 1 0 1	(–5)	1 0 1 0
+ (+2)	+ 0 0 1 0	+ (+2)	+ 0 0 1 0
(+7)	0 1 1 1	(–3)	1 1 0 0
(+5)	0 1 0 1	(–5)	1 0 1 0
+ (–2)	+ 1 1 0 1	+ (–2)	+ 1 1 0 1
(+3)	1 0 0 1 0	(–7)	1 0 1 1 1
	→ 1		→ 1
	0 0 1 1		1 0 0 0

Figure 3.8 Examples of 1's complement addition.

result represent the number 2 rather than 3, which is a wrong result. Interestingly, if we take the carry-out from the sign-bit position and add it to the result in the least-significant bit position, the new result is the correct sum of 3. This correction is indicated in blue in the figure. A similar situation arises when adding $(-5) + (-2) = (-7)$. After the initial addition the result is wrong because the four bits of the sum are 0111, which represents $+7$ rather than -7. But again, there is a carry-out from the sign-bit position, which can be used to correct the result by adding it in the LSB position, as shown in Figure 3.8.

The conclusion from these examples is that the addition of 1's complement numbers may or may not be simple. In some cases a correction is needed, which amounts to an extra addition that must be performed. Consequently, the time needed to add two 1's complement numbers may be twice as long as the time needed to add two unsigned numbers.

2's Complement Addition

Consider the same combinations of numbers as used in the 1's complement example. Figure 3.9 indicates how the addition is performed using 2's complement numbers. Adding $5 + 2 = 7$ and $(-5) + 2 = (-3)$ is straightforward. The computation $5 + (-2) = 3$ generates the correct four bits of the result, namely 0011. There is a carry-out from the sign-bit position, which we can simply ignore. The fourth case is $(-5) + (-2) = (-7)$. Again, the four bits of the result, 1001, give the correct sum (-7). In this case also, the carry-out from the sign-bit position can be ignored.

As illustrated by these examples, the addition of 2's complement numbers is very simple. When the numbers are added, the result is always correct. If there is a carry-out from the sign-bit position, it is simply ignored. Therefore, the addition process is the same, regardless of the signs of the operands. It can be performed by an adder circuit, such as the one shown in Figure 3.5. Hence the 2's complement notation is highly suitable for the implementation of addition operations. We will now consider its use in subtraction operations.

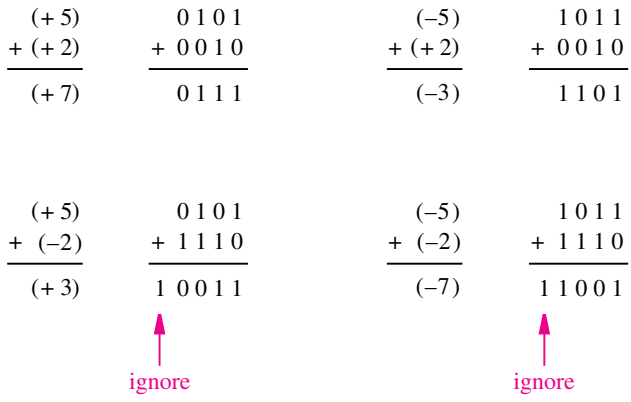


Figure 3.9 Examples of 2's complement addition.

2's Complement Subtraction

The easiest way of performing subtraction is to negate the subtrahend and add it to the minuend. This is done by finding the 2's complement of the subtrahend and then performing the addition. Figure 3.10 illustrates the process. The operation $5 - (+2) = 3$ involves finding the 2's complement of +2, which is 1110. When this number is added to 0101, the result is $0011 = (+3)$ and a carry-out from the sign-bit position occurs, which is ignored. A similar situation arises for $(-5) - (+2) = (-7)$. In the remaining two cases there is no carry-out, and the result is correct.

As a graphical aid to visualize the addition and subtraction examples in Figures 3.9 and 3.10, we can place all possible four-bit patterns on a modulo-16 circle given in Figure 3.11*a*. If these bit patterns represented unsigned integers, they would be numbers 0 to 15. If they represent 2's-complement integers, then the numbers range from −8 to +7, as shown. The addition operation is done by stepping in the clockwise direction by the magnitude of the number to be added. For example, $-5 + 2$ is determined by starting at 1011 (= −5) and moving clockwise two steps, giving the result 1101 (= −3). Figure 3.11*b* shows how subtraction can be performed on the modulo-16 circle, using the example $5 - 2 = 3$. We can start at 0101 (= +5) and move counterclockwise by two steps, which gives 0011 (= +3). But, we can also use the 2's complement of 2 and add this value by stepping in the clockwise

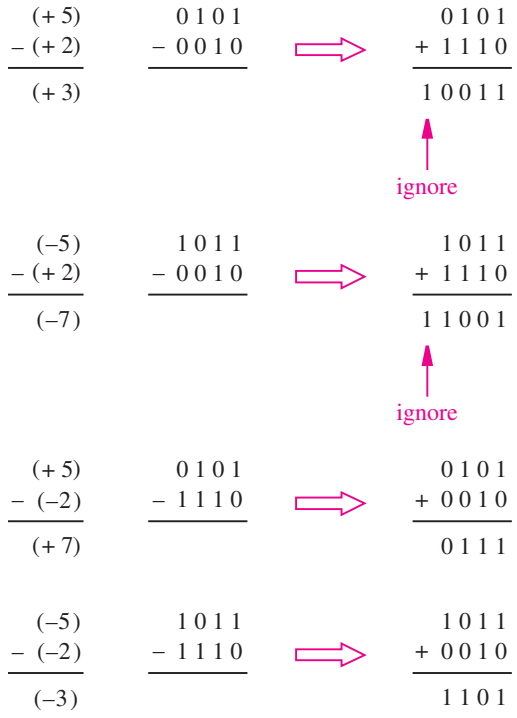
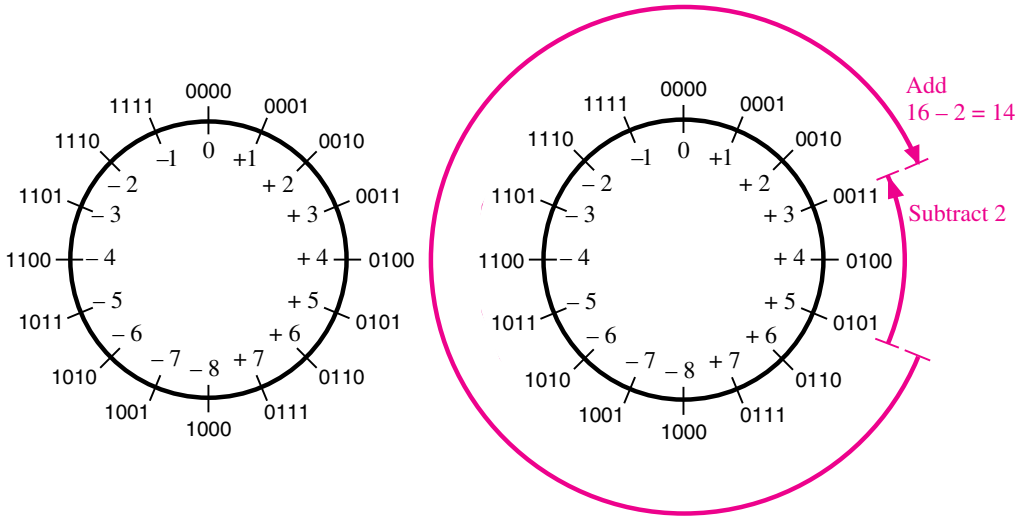


Figure 3.10 Examples of 2's complement subtraction.



(a) The number circle (b) Subtracting 2 by adding its 2's complement

Figure 3.11 Graphical interpretation of four-bit 2's complement numbers.

direction as shown in the figure. Since there are 16 numbers on the circle, the value we need to add is $16 - 2 = (14)_{10} = (1110)_2$.

The key conclusion of this section is that the subtraction operation can be realized as the addition operation, using a 2's complement of the subtrahend, regardless of the signs of the two operands. Therefore, it should be possible to use the same adder circuit to perform both addition and subtraction.

3.3.3 ADDER AND SUBTRACTOR UNIT

The only difference between performing addition and subtraction is that for subtraction it is necessary to use the 2's complement of one operand. Let X and Y be the two operands, such that Y serves as the subtrahend in subtraction. From Section 3.3.1 we know that a 2's complement can be obtained by adding 1 to the 1's complement of Y. Adding 1 in the least-significant bit position can be accomplished simply by setting the carry-in bit c_0 to 1. A 1's complement of a number is obtained by complementing each of its bits. This could be done with NOT gates, but we need a more flexible circuit where we can use the true value of Y for addition and its complement for subtraction.

In Section 3.2 we explained that two-input XOR gates can be used to choose between true and complemented versions of an input value, under the control of the other input. This idea can be applied in the design of the adder/subtractor unit as follows. Assume that there exists a control signal that chooses whether addition or subtraction is to be performed. Let

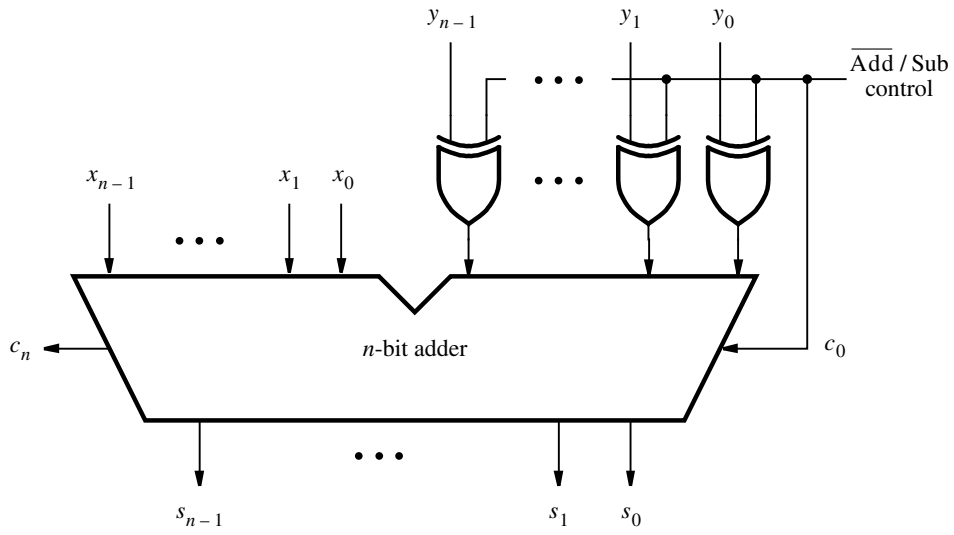


Figure 3.12 Adder/subtractor unit.

this signal be called Add/Sub. Also, let its value be 0 for addition and 1 for subtraction. To indicate this fact, we placed a bar over Add. This is a commonly used convention, where a bar over a name means that the action specified by the name is to be taken if the control signal has the value 0. Now let each bit of Y be connected to one input of an XOR gate, with the other input connected to Add/Sub. The outputs of the XOR gates represent Y if Add/Sub $= 0$, and they represent the 1's complement of Y if Add/Sub $= 1$. This leads to the circuit in Figure 3.12. The main part of the circuit is an n-bit adder, which can be implemented using the ripple-carry structure of Figure 3.5. Note that the control signal Add/Sub is also connected to the carry-in c_0. This makes $c_0 = 1$ when subtraction is to be performed, thus adding the 1 that is needed to form the 2's complement of Y. When the addition operation is performed, we will have $c_0 = 0$.

The combined adder/subtractor unit is a good example of an important concept in the design of logic circuits. It is useful to design circuits to be as flexible as possible and to exploit common portions of circuits for as many tasks as possible. This approach minimizes the number of gates needed to implement such circuits, and it reduces the wiring complexity substantially.

3.3.4 RADIX-COMPLEMENT SCHEMES*

The 2's complement scheme is just a special case of radix-complement schemes which we discuss in this section. This general discussion can be skipped without loss of continuity in the context of computer technology.

The idea of performing a subtraction operation by addition of a complement of the subtrahend is not restricted to binary numbers. We can gain some insight into the workings of the 2's complement scheme by considering its counterpart in the decimal number system. Consider the subtraction of two-digit decimal numbers. Computing a result such as $74 - 33 = 41$ is simple because each digit of the subtrahend is smaller than the corresponding digit of the minuend; therefore, no borrow is needed in the computation. But computing $74 - 36 = 38$ is not as simple because a borrow is needed in subtracting the least-significant digit. If a borrow occurs, the computation becomes more complicated.

Suppose that we restructure the required computation as follows

$$\begin{aligned} 74 - 36 &= 74 + 100 - 100 - 36 \\ &= 74 + (100 - 36) - 100 \end{aligned}$$

Now two subtractions are needed. Subtracting 36 from 100 still involves borrows. But noting that $100 = 99 + 1$, these borrows can be avoided by writing

$$\begin{aligned} 74 - 36 &= 74 + (99 + 1 - 36) - 100 \\ &= 74 + (99 - 36) + 1 - 100 \end{aligned}$$

The subtraction in parentheses does not require borrows; it is performed by subtracting each digit of the subtrahend from 9. We can see a direct correlation between this expression and the one used for 2's complement, as reflected in the circuit in Figure 3.12. The operation $(99 - 36)$ is analogous to complementing the subtrahend Y to find its 1's complement, which is the same as subtracting each bit from 1. Using decimal numbers, we find the *9's complement* of the subtrahend by subtracting each digit from 9. In Figure 3.12 we add the carry-in of 1 to form the 2's complement of Y. In our decimal example we perform $(99 - 36) + 1 = 64$. Here 64 is the 10's complement of 36. For an n-digit decimal number, N, its *10's complement*, K_{10}, is defined as $K_{10} = 10^n - N$, while its 9's complement, K_9, is $K_9 = (10^n - 1) - N$.

Thus the required subtraction $(74 - 36)$ can be performed by addition of the 10's complement of the subtrahend, as in

$$\begin{aligned} 74 - 36 &= 74 + 64 - 100 \\ &= 138 - 100 \\ &= 38 \end{aligned}$$

The subtraction $138 - 100$ is trivial because it means that the leading digit in 138 is simply deleted. This is analogous to ignoring the carry-out from the circuit in Figure 3.12, as discussed for the subtraction examples in Figure 3.10.

Example 3.1 Suppose that A and B are n-digit decimal numbers. Using the above 10's-complement approach, B can be subtracted from A as follows:

$$A - B = A + (10^n - B) - 10^n$$

If $A \geq B$, then the operation $A + (10^n - B)$ produces a carry-out of 1. This carry is equivalent to 10^n; hence it can be simply ignored.

But if $A < B$, then the operation $A + (10^n - B)$ produces a carry-out of 0. Let the result obtained be M, so that

$$A - B = M - 10^n$$

We can rewrite this as

$$10^n - (B - A) = M$$

The left side of this equation is the 10's complement of $(B - A)$. The 10's complement of a positive number represents a negative number that has the same magnitude. Hence M correctly represents the negative value obtained from the computation $A - B$ when $A < B$. This concept is illustrated in the examples that follow.

Example 3.2

When dealing with binary signed numbers we use 0 in the left-most bit position to denote a positive number and 1 to denote a negative number. If we wanted to build hardware that operates on signed decimal numbers, we could use a similar approach. Let 0 in the left-most digit position denote a positive number and let 9 denote a negative number. Note that 9 is the 9's complement of 0 in the decimal system, just as 1 is the 1's complement of 0 in the binary system.

Thus, using three-digit signed numbers, $A = 045$ and $B = 027$ are positive numbers with magnitudes 45 and 27, respectively. The number B can be subtracted from A as follows

$$\begin{aligned} A - B &= 045 - 027 \\ &= 045 + 1000 - 1000 - 027 \\ &= 045 + (999 - 027) + 1 - 1000 \\ &= 045 + 972 + 1 - 1000 \\ &= 1018 - 1000 \\ &= 018 \end{aligned}$$

This gives the correct answer of +18.

Next consider the case where the minuend has lower value than the subtrahend. This is illustrated by the computation

$$\begin{aligned} B - A &= 027 - 045 \\ &= 027 + 1000 - 1000 - 045 \\ &= 027 + (999 - 045) + 1 - 1000 \\ &= 027 + 954 + 1 - 1000 \\ &= 982 - 1000 \end{aligned}$$

From this expression it appears that we still need to perform the subtraction $982 - 1000$. But as seen in Example 3.1, this can be rewritten as

$$\begin{aligned} 982 &= 1000 + B - A \\ &= 1000 - (A - B) \end{aligned}$$

Therefore, 982 is the negative number that results when forming the 10's complement of $(A - B)$. From the previous computation we know that $(A - B) = 018$, which denotes $+18$. Thus the signed number 982 is the 10's complement representation of -18, which is the required result.

These examples illustrate that signed numbers can be subtracted without using a subtraction operation that involves borrows. The only subtraction needed is in forming the 9's complement of the subtrahend, in which case each digit is simply subtracted from 9. Thus a circuit that forms the 9's complement, combined with a normal adder circuit, will suffice for both addition and subtraction of decimal signed numbers. A key point is that the hardware needs to deal only with n digits if n-digit numbers are used. Any carry that may be generated from the left-most digit position is simply ignored.

The concept of subtracting a number by adding its radix-complement is general. If the radix is r, then the r's complement, K_r, of an n-digit number, N, is determined as $K_r = r^n - N$. The $(r - 1)$'s complement, K_{r-1}, is defined as $K_{r-1} = (r^n - 1) - N$; it is computed simply by subtracting each digit of N from the value $(r - 1)$. The $(r - 1)$'s complement is referred to as the *diminished-radix complement*. Circuits for forming the $(r - 1)$'s complements are simpler than those for general subtraction that involves borrows. The circuits are particularly simple in the binary case, where the 1's complement requires just inverting each bit.

Example 3.3 In Figure 3.10 we illustrated the subtraction operation on binary numbers given in 2's-complement representation. Consider the computation $(+5) - (+2) = (+3)$, using the approach discussed above. Each number is represented by a four-bit pattern. The value 2^4 is represented as 10000. Then

$$\begin{aligned} 0101 - 0010 &= 0101 + (10000 - 0010) - 10000 \\ &= 0101 + (1111 - 0010) + 1 - 10000 \\ &= 0101 + 1101 + 1 - 10000 \\ &= 10011 - 10000 \\ &= 0011 \end{aligned}$$

Because $5 > 2$, there is a carry from the fourth bit position. It represents the value 2^4, denoted by the pattern 10000.

Example 3.4 Consider now the computation $(+2) - (+5) = (-3)$, which gives

$$\begin{aligned} 0010 - 0101 &= 0010 + (10000 - 0101) - 10000 \\ &= 0010 + (1111 - 0101) + 1 - 10000 \\ &= 0010 + 1010 + 1 - 10000 \\ &= 1101 - 10000 \end{aligned}$$

Because $2 < 5$, there is no carry from the fourth bit position. The answer, 1101, is the 2's-complement representation of -3. Note that

$$\begin{aligned} 1101 &= 10000 + 0010 - 0101 \\ &= 10000 - (0101 - 0010) \\ &= 10000 - 0011 \end{aligned}$$

indicating that 1101 is the 2's complement of 0011 (+3).

Example 3.5

Finally, consider the case where the subtrahend is a negative number. The computation $(+5) - (-2) = (+7)$ is done as follows

$$\begin{aligned} 0101 - 1110 &= 0101 + (10000 - 1110) - 10000 \\ &= 0101 + (1111 - 1110) + 1 - 10000 \\ &= 0101 + 0001 + 1 - 10000 \\ &= 0111 - 10000 \end{aligned}$$

While $5 > (-2)$, the pattern 1110 is greater than the pattern 0101 when the patterns are treated as unsigned numbers. Therefore, there is no carry from the fourth bit position. The answer 0111 is the 2's complement representation of +7. Note that

$$\begin{aligned} 0111 &= 10000 + 0101 - 1110 \\ &= 10000 - (1110 - 0101) \\ &= 10000 - 1001 \end{aligned}$$

and 1001 represents -7.

3.3.5 ARITHMETIC OVERFLOW

The result of addition or subtraction is supposed to fit within the significant bits used to represent the numbers. If n bits are used to represent signed numbers, then the result must be in the range -2^{n-1} to $2^{n-1} - 1$. If the result does not fit in this range, then we say that *arithmetic overflow* has occurred. To ensure the correct operation of an arithmetic circuit, it is important to be able to detect the occurrence of overflow.

Figure 3.13 presents the four cases where 2's-complement numbers with magnitudes of 7 and 2 are added. Because we are using four-bit numbers, there are three significant bits, b_{2-0}. When the numbers have opposite signs, there is no overflow. But if both numbers have the same sign, the magnitude of the result is 9, which cannot be represented with just three significant bits; therefore, overflow occurs. The key to determining whether overflow occurs is the carry-out from the MSB position, called c_3 in the figure, and from the sign-bit position, called c_4. The figure indicates that overflow occurs when these carry-outs have different values, and a correct sum is produced when they have the same value. Indeed, this is true in general for both addition and subtraction of 2's-complement numbers. As a quick

(+7)	0 1 1 1	(–7)	1 0 0 1
+ (+2)	+ 0 0 1 0	+ (+2)	+ 0 0 1 0
(+9)	1 0 0 1	(–5)	1 0 1 1
	$c_4 = 0$		$c_4 = 0$
	$c_3 = 1$		$c_3 = 0$
(+7)	0 1 1 1	(–7)	1 0 0 1
+ (–2)	+ 1 1 1 0	+ (–2)	+ 1 1 1 0
(+5)	1 0 1 0 1	(–9)	1 0 1 1 1
	$c_4 = 1$		$c_4 = 1$
	$c_3 = 1$		$c_3 = 0$

Figure 3.13 Examples for determination of overflow.

check of this statement, consider the examples in Figure 3.9 where the numbers are small enough so that overflow does not occur in any case. In the top two examples in the figure, there is a carry-out of 0 from both sign and MSB positions. In the bottom two examples, there is a carry-out of 1 from both positions. Therefore, for the examples in Figures 3.9 and 3.13, the occurrence of overflow is detected by

$$\begin{aligned}\text{Overflow} &= c_3\overline{c}_4 + \overline{c}_3 c_4 \\ &= c_3 \oplus c_4\end{aligned}$$

For n-bit numbers we have

$$\text{Overflow} = c_{n-1} \oplus c_n$$

Thus the circuit in Figure 3.12 can be modified to include overflow checking with the addition of one XOR gate.

An alternative and more intuitive way of detecting the arithmetic overflow is to observe that overflow occurs if both summands have the same sign but the resulting sum has a different sign. Let $X = x_3x_2x_1x_0$ and $Y = y_3y_2y_1y_0$ represent four-bit 2's-complement numbers, and let $S = s_3s_2s_1s_0$ be the sum $S = X + Y$. Then

$$\text{Overflow} = x_3y_3\overline{s}_3 + \overline{x}_3\overline{y}_3 s_3$$

The carry-out and overflow signals indicate whether the result of a given addition is too large to fit into the number of bits available to represent the sum. The carry-out is meaningful only when unsigned numbers are involved, while the overflow is meaningful only in the case of signed numbers. In a typical computer, it is prudent to use the same adder circuits for dealing with both unsigned and signed operands, thus reducing the amount of circuitry required. This means that both the carry-out and overflow signals should be generated, as

we have discussed. Then, a program instruction that specifies unsigned operands can use the carry-out signal, while an instruction that has signed operands can use the overflow signal.

3.3.6 PERFORMANCE ISSUES

When buying a digital system, such as a computer, the buyer pays particular attention to the performance that the system is expected to provide and to the cost of acquiring the system. Superior performance usually comes at a higher cost. However, a large increase in performance can often be achieved at a modest increase in cost. A commonly used indicator of the value of a system is its *price/performance ratio*.

The addition and subtraction of numbers are fundamental operations that are performed frequently in the course of a computation. The speed with which these operations are performed has a strong impact on the overall performance of a computer. In light of this, let us take a closer look at the speed of the adder/subtractor unit in Figure 3.12. We are interested in the largest delay from the time the operands X and Y are presented as inputs, until the time all bits of the sum S and the final carry-out, c_n, are valid. Most of this delay is caused by the n-bit adder circuit. Assume that the adder is implemented using the ripple-carry structure in Figure 3.5 and that each full-adder stage is the circuit in Figure 3.3*c*. The delay for the carry-out signal in this circuit, Δt, is equal to two gate delays. From Section 3.2.2 we know that the final result of the addition will be valid after a delay of $n\Delta t$, which is equal to $2n$ gate delays. In addition to the delay in the ripple-carry path, there is also a delay in the XOR gates that feed either the true or complemented value of Y to the adder inputs. If this delay is equal to one gate delay, then the total delay of the circuit in Figure 3.12 is $2n + 1$ gate delays. For a large n, say $n = 32$ or $n = 64$, the delay would lead to unacceptably poor performance. Therefore, it is important to find faster circuits to perform addition.

The speed of any circuit is limited by the longest delay along the paths through the circuit. In the case of the circuit in Figure 3.12, the longest delay is along the path from the y_i input, through the XOR gate and through the carry circuit of each adder stage. The longest delay is often referred to as the *critical-path delay*, and the path that causes this delay is called the *critical path*.

3.4 FAST ADDERS

The performance of a large digital system is dependent on the speed of circuits that form its various functional units. Obviously, better performance can be achieved using faster circuits. This can be accomplished by using superior (usually newer) technology in which the delays in basic gates are reduced. But it can also be accomplished by changing the overall structure of a functional unit, which may lead to even more impressive improvement. In this section we will discuss an alternative for implementation of an n-bit adder, which substantially reduces the time needed to add numbers.

3.4.1 CARRY-LOOKAHEAD ADDER

To reduce the delay caused by the effect of carry propagation through the ripple-carry adder, we can attempt to evaluate quickly for each stage whether the carry-in from the previous stage will have a value 0 or 1. If a correct evaluation can be made in a relatively short time, then the performance of the complete adder will be improved.

From Figure 3.3*b* the carry-out function for stage i can be realized as

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

If we factor this expression as

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

then it can be written as

$$c_{i+1} = g_i + p_i c_i \qquad \textbf{[3.3]}$$

where

$$g_i = x_i y_i$$
$$p_i = x_i + y_i$$

The function g_i is equal to 1 when both inputs x_i and y_i are equal to 1, regardless of the value of the incoming carry to this stage, c_i. Since in this case stage i is guaranteed to generate a carry-out, g is called the *generate* function. The function p_i is equal to 1 when at least one of the inputs x_i and y_i is equal to 1. In this case a carry-out is produced if $c_i = 1$. The effect is that the carry-in of 1 is propagated through stage i; hence p_i is called the *propagate* function.

Expanding the expression 3.3 in terms of stage $i - 1$ gives

$$\begin{aligned} c_{i+1} &= g_i + p_i(g_{i-1} + p_{i-1}c_{i-1}) \\ &= g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-1} \end{aligned}$$

The same expansion for other stages, ending with stage 0, gives

$$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \cdots p_2 p_1 g_0 + p_i p_{i-1} \cdots p_1 p_0 c_0 \qquad \textbf{[3.4]}$$

This expression represents a two-level AND-OR circuit in which c_{i+1} is evaluated very quickly. An adder based on this expression is called a *carry-lookahead adder*.

To appreciate the physical meaning of expression 3.4, it is instructive to consider its effect on the construction of a fast adder in comparison with the details of the ripple-carry adder. We will do so by examining the detailed structure of the two stages that add the least-significant bits, namely, stages 0 and 1. Figure 3.14 shows the first two stages of a ripple-carry adder in which the carry-out functions are implemented as indicated in expression 3.3. Each stage is essentially the circuit from Figure 3.3*c* except that an extra OR gate is used (which produces the p_i signal), instead of an AND gate because we factored the sum-of-products expression for c_{i+1}.

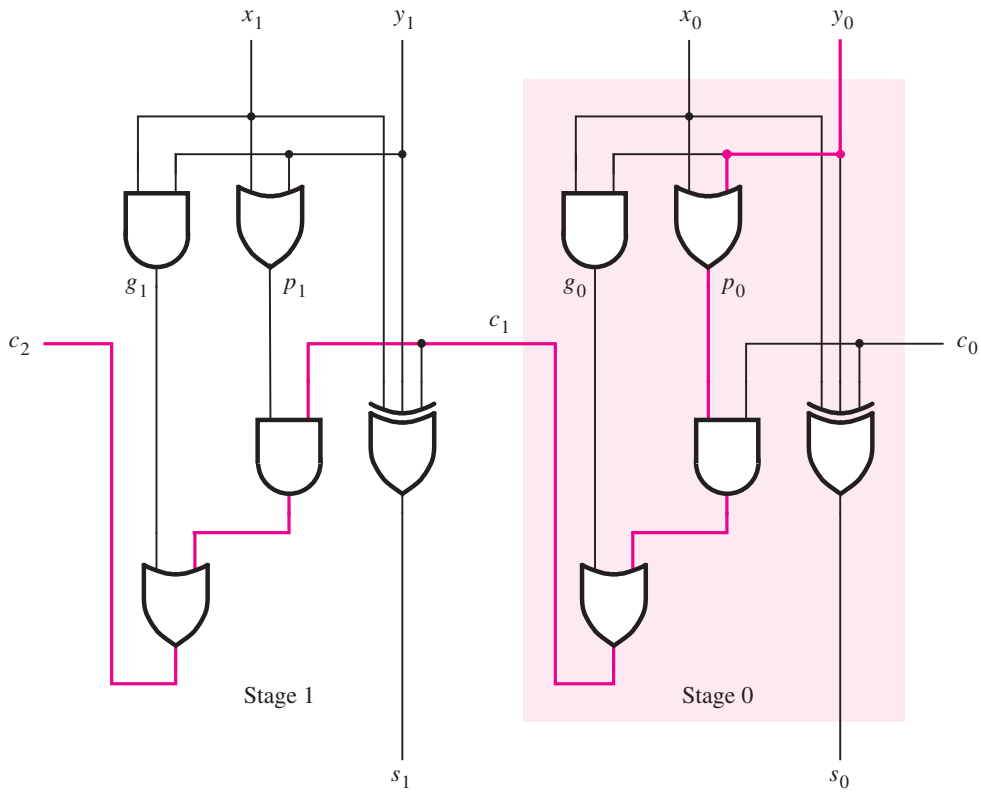


Figure 3.14 A ripple-carry adder based on expression 3.3.

The slow speed of the ripple-carry adder is caused by the long path along which a carry signal must propagate. In Figure 3.14 the critical path is from inputs x_0 and y_0 to the output c_2. It passes through five gates, as highlighted in blue. The path in other stages of an n-bit adder is the same as in stage 1. Therefore, the total number of gate delays along the critical path is $2n + 1$.

Figure 3.15 gives the first two stages of the carry-lookahead adder, using expression 3.4 to implement the carry-out functions. Thus

$$c_1 = g_0 + p_0 c_0$$
$$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$$

This circuit does not have the long ripple-carry path that is present in Figure 3.14. Instead, all carry signals are produced after three gate delays: one gate delay is needed to produce the generate and propagate signals g_0, g_1, p_0, and p_1, and two more gate delays are needed to produce c_1 and c_2 concurrently. Extending the circuit to n bits, the final carry-out signal

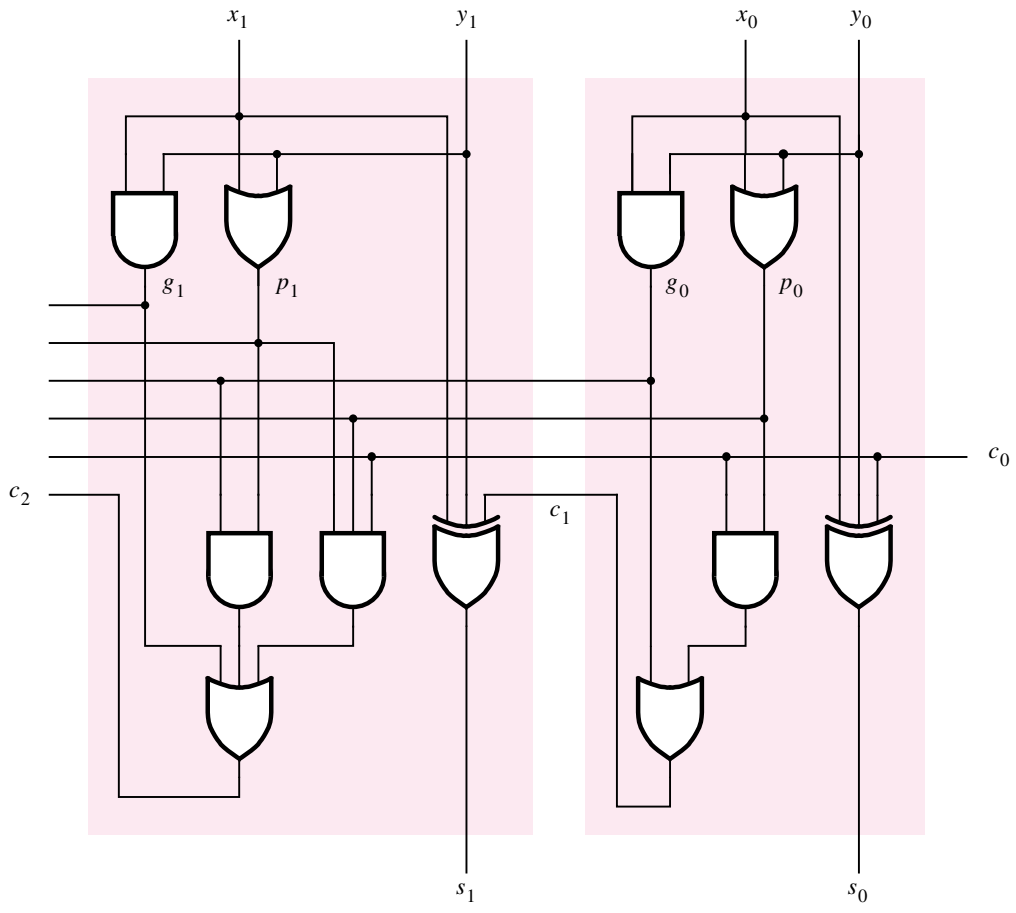


Figure 3.15 The first two stages of a carry-lookahead adder.

c_n would also be produced after only three gate delays because expression 3.4 is just a large two-level (AND-OR) circuit.

The total delay in the n-bit carry-lookahead adder is four gate delays. The values of all g_i and p_i signals are determined after one gate delay. It takes two more gate delays to evaluate all carry signals. Finally, it takes one more gate delay (XOR) to generate all sum bits. The key to the good performance of the adder is quick evaluation of carry signals.

The complexity of an n-bit carry-lookahead adder increases rapidly as n becomes larger. To reduce the complexity, we can use a *hierarchical* approach in designing large adders. Suppose that we want to design a 32-bit adder. We can divide this adder into 4 eight-bit blocks, such that block 0 adds bits 7 ... 0, block 1 adds bits 15 ... 8, block 2 adds bits 23 ... 16, and block 3 adds bits 31 ... 24. Then we can implement each block as an eight-bit carry-lookahead adder. The carry-out signals from the four blocks are c_8, c_{16}, c_{24},

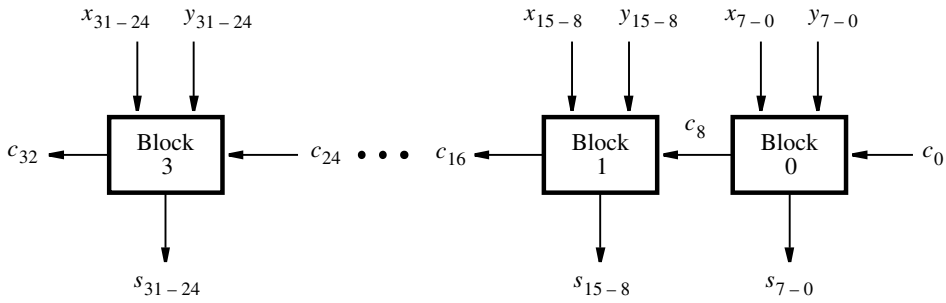


Figure 3.16 A hierarchical carry-lookahead adder with ripple-carry between blocks.

and c_{32}. Now we have two possibilities. We can connect the four blocks as four stages in a ripple-carry adder. Thus while carry-lookahead is used within each block, the carries ripple between the blocks. This circuit is illustrated in Figure 3.16.

Instead of using a ripple-carry approach between blocks, a faster circuit can be designed in which a second-level carry-lookahead is performed to produce quickly the carry signals between blocks. The structure of this "hierarchical carry-lookahead adder" is shown in Figure 3.17. Each block in the top row includes an eight-bit carry-lookahead adder, based on generate and propagate signals for each stage in the block, as discussed before. However, instead of producing a carry-out signal from the most-significant bit of the block, each block produces generate and propagate signals for the entire block. Let G_j and P_j denote these signals for each block j. Now G_j and P_j can be used as inputs to a second-level carry-lookahead circuit at the bottom of Figure 3.17, which evaluates all carries between blocks. We can derive the block generate and propagate signals for block 0 by examining the expression for c_8

$$\begin{aligned} c_8 = {} & g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4 + p_7p_6p_5p_4g_3 + p_7p_6p_5p_4p_3g_2 \\ & + p_7p_6p_5p_4p_3p_2g_1 + p_7p_6p_5p_4p_3p_2p_1g_0 + p_7p_6p_5p_4p_3p_2p_1p_0c_0 \end{aligned}$$

The last term in this expression specifies that, if all eight propagate functions are 1, then the carry-in c_0 is propagated through the entire block. Hence

$$P_0 = p_7p_6p_5p_4p_3p_2p_1p_0$$

The rest of the terms in the expression for c_8 represent all other cases when the block produces a carry-out. Thus

$$G_0 = g_7 + p_7g_6 + p_7p_6g_5 + \cdots + p_7p_6p_5p_4p_3p_2p_1g_0$$

The expression for c_8 in the hierarchical adder is given by

$$c_8 = G_0 + P_0c_0$$

For block 1 the expressions for G_1 and P_1 have the same form as for G_0 and P_0 except that each subscript i is replaced by $i + 8$. The expressions for G_2, P_2, G_3, and P_3 are derived in

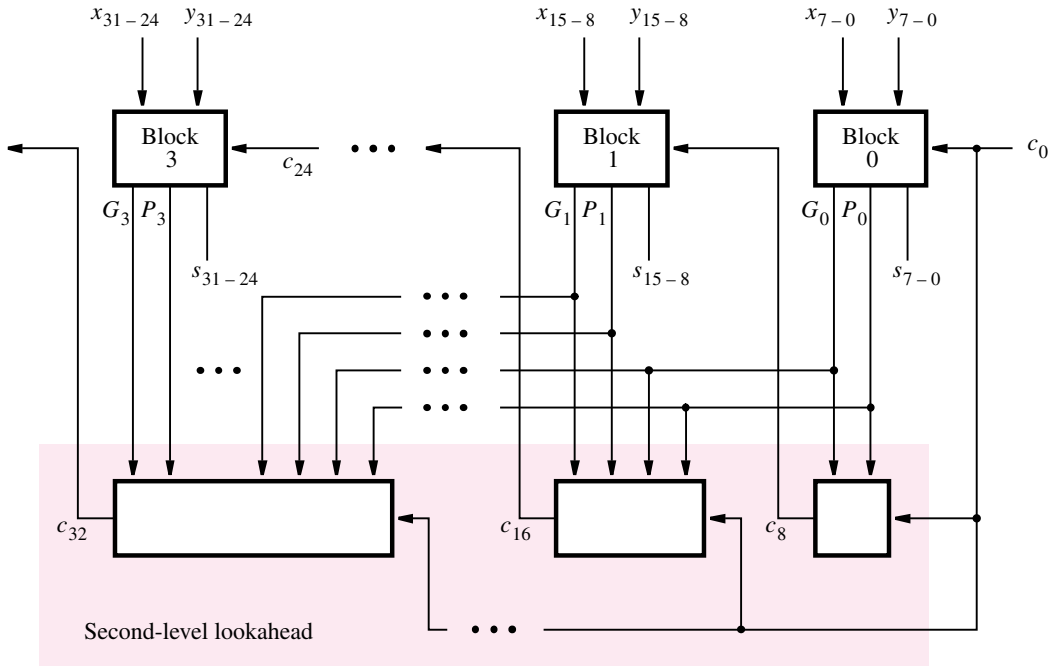


Figure 3.17 A hierarchical carry-lookahead adder.

the same way. The expression for the carry-out of block 1, c_{16}, is

$$\begin{aligned} c_{16} &= G_1 + P_1 c_8 \\ &= G_1 + P_1 G_0 + P_1 P_0 c_0 \end{aligned}$$

Similarly, the expressions for c_{24} and c_{32} are

$$\begin{aligned} c_{24} &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0 \\ c_{32} &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0 \end{aligned}$$

Using this scheme, it takes two more gate delays to produce the carry signals c_8, c_{16}, c_{24}, and c_{32} than the time needed to generate the G_j and P_j functions. Therefore, since G_j and P_j require three gate delays, c_8, c_{16}, c_{24}, and c_{32} are available after five gate delays. The time needed to add two 32-bit numbers involves these five gate delays plus two more to produce the internal carries in blocks 1, 2, and 3, plus one more gate delay (XOR) to generate each sum bit. This gives a total of eight gate delays.

In Section 3.3.6 we determined that it takes $2n + 1$ gate delays to add two numbers using a ripple-carry adder. For 32-bit numbers this implies 65 gate delays. It is clear that the carry-lookahead adder offers a large performance improvement. The trade-off is much greater complexity of the required circuit.

Technology Considerations

The preceding delay analysis assumes that gates with any number of inputs can be used. But, the number of gate inputs, referred to as the *fan-in* of the gate, has to be limited in practice as we discuss in Appendix B. Therefore the reality of fan-in constraints must be taken into account. To illustrate this problem, consider the expressions for the first eight carries:

$$\begin{aligned}
c_1 &= g_0 + p_0c_0 \\
c_2 &= g_1 + p_1g_0 + p_1p_0c_0 \\
&\vdots \\
c_8 &= g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4 + p_7p_6p_5p_4g_3 + p_7p_6p_5p_4p_3g_2 \\
&\quad + p_7p_6p_5p_4p_3p_2g_1 + p_7p_6p_5p_4p_3p_2p_1g_0 + p_7p_6p_5p_4p_3p_2p_1p_0c_0
\end{aligned}$$

Suppose that the maximum fan-in of the gates is four inputs. Then it is impossible to implement all of these expressions with a two-level AND-OR circuit. The biggest problem is c_8, where one of the AND gates requires nine inputs; moreover, the OR gate also requires nine inputs. To meet the fan-in constraint, we can rewrite the expression for c_8 as

$$\begin{aligned}
c_8 = {} & (g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4) + [(p_7p_6p_5p_4)(g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0)] \\
& + (p_7p_6p_5p_4)(p_3p_2p_1p_0)c_0
\end{aligned}$$

To implement this expression we need ten AND gates and three OR gates. The propagation delay in generating c_8 consists of one gate delay to develop all g_i and p_i, two gate delays to produce the sum-of-products terms in parentheses, one gate delay to form the product term in square brackets, and one delay for the final ORing of terms. Hence c_8 is valid after five gate delays, rather than the three gate delays that would be needed without the fan-in constraint.

Because fan-in limitations reduce the speed of the carry-lookahead adder, some devices that are characterized by low fan-in include dedicated circuitry for implementation of fast adders. Examples of such devices include FPGAs, which are described in Appendix B.

3.5 DESIGN OF ARITHMETIC CIRCUITS USING CAD TOOLS

In this section we show how the arithmetic circuits can be designed by using CAD tools.

3.5.1 DESIGN OF ARITHMETIC CIRCUITS USING SCHEMATIC CAPTURE

An obvious way to design an arithmetic circuit via schematic capture is to draw a schematic that contains the necessary logic gates. For example, to create an n-bit adder, we could first draw a schematic that represents a full-adder. Then an n-bit ripple-carry adder could be

created by drawing a higher-level schematic that connects together *n instances* of the full-adder. A hierarchical schematic created in this manner would look like the circuit shown in Figure 3.5. We could also use this methodology to create an adder/subtractor circuit, such as the circuit depicted in Figure 3.12.

The main problem with this approach is that it is cumbersome, especially when the number of bits is large. This issue is even more apparent if we consider creating a schematic for a carry-lookahead adder. As shown in Section 3.4.1, the carry circuitry in each stage of the carry-lookahead adder becomes increasingly more complex. Hence it is necessary to draw a separate schematic for each stage of the adder. A better approach for creating arithmetic circuits via schematic capture is to use predefined subcircuits.

The schematic capture tools provide a library of graphical symbols that represent basic logic gates. These gates are used to create schematics of relatively simple circuits. In addition to basic gates, most schematic capture tools also provide a library of commonly-used circuits, such as adders. Each circuit is provided as a module that can be imported into a schematic and used as part of a larger circuit.

Different vendors of CAD tools have their specific approaches to how schematic capture is performed. Instead of dealing with the schematic capture approach, we will concentrate on the more convenient and flexible approach of using Verilog to design our circuits.

3.5.2 DESIGN OF ARITHMETIC CIRCUITS USING VERILOG

We said in Section 3.5.1 that an obvious way to create an n-bit adder is to draw a hierarchical schematic that contains n full-adders. This approach can also be followed by using Verilog, by first creating a Verilog module for a full-adder and then defining a higher-level module that uses *n instances* of the full-adder. As a first attempt at designing arithmetic circuits by using Verilog, we will show how to write the hierarchical code for a ripple-carry adder. Hierarchical Verilog code was introduced in Chapter 2.

Suppose that we wish to implement the full-adder circuit given in Figure 3.3*c*, which has the inputs *Cin*, *x*, and *y*, and produces the outputs *s* and *Cout*. One way of specifying this circuit in Verilog is to use the gate-level primitives as shown in Figure 3.18. Each of the three AND gates in the circuit is defined by a separate statement. Verilog allows combining such statements into a single statement as shown in Figure 3.19. In this case, commas are used to separate the definition of each AND gate.

Another possibility is to use functional expressions as indicated in Figure 3.20. The XOR operation is denoted by the ^ sign. Again, it is possible to combine the two continuous assignment statements into a single statement as shown in Figure 3.21.

Both of the above approaches result in the same full-adder circuit being synthesized. We can now create a separate Verilog module for the ripple-carry adder, which instantiates the *fulladd* module as a subcircuit. One method of doing this is shown in Figure 3.22. The module comprises the code for a four-bit ripple-carry adder, named *adder4*. One of the four-bit numbers to be added is represented by the four signals x_3, x_2, x_1, x_0, and the other number is represented by y_3, y_2, y_1, y_0. The sum is represented by s_3, s_2, s_1, s_0. The circuit incorporates a carry input, *carryin*, into the least-significant bit position and a carry output, *carryout*, from the most-significant bit position.

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   xor (s, x, y, Cin);
   and (z1, x, y);
   and (z2, x, Cin);
   and (z3, y, Cin);
   or (Cout, z1, z2, z3);

endmodule
```

Figure 3.18 Verilog code for the full-adder using gate-level primitives.

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   xor (s, x, y, Cin);
   and (z1, x, y),
       (z2, x, Cin),
       (z3, y, Cin);
   or (Cout, z1, z2, z3);

endmodule
```

Figure 3.19 Another version of Verilog code from Figure 3.18.

The four-bit adder in Figure 3.22 is described using four *instantiation* statements. Each statement begins with the name of the module, *fulladd*, that is being instantiated, followed by an *instance name*. The instance names must be unique. The least-significant stage in the adder is named *stage0* and the most-significant stage is *stage3*. The signal names in the *adder4* module that are to be connected to each input and output port on the *fulladd* module are then listed. These signals are listed in the same order as in the *fulladd* module, namely the order *Cin*, *x*, *y*, *s*, *Cout*.

As discussed in Section 2.10, the signal names associated with each instance of the *fulladd* module implicitly specify how the full-adders are connected together. For example, the carry-out of the *stage0* instance is connected to the carry-in of the *stage1* instance. The synthesized circuit has the same structure as the one shown in Figure 3.5. The *fulladd* module may be included in the same Verilog source code file as the *adder4* module, as we have done in Figure 3.22, but it may also comprise a separate file. In the latter case, the location of the file *fulladd* has to be indicated to the compiler.

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign s = x ^ y ^ Cin;
   assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

Figure 3.20 Verilog code for the full-adder using continuous assignment.

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign s = x ^ y ^ Cin,
          Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

Figure 3.21 Another version of Verilog code from Figure 3.20.

```
module adder4 (carryin, x3, x2, x1, x0, y3, y2, y1, y0, s3, s2, s1, s0, carryout);
   input carryin, x3, x2, x1, x0, y3, y2, y1, y0;
   output s3, s2, s1, s0, carryout;

   fulladd stage0 (carryin, x0, y0, s0, c1);
   fulladd stage1 (c1, x1, y1, s1, c2);
   fulladd stage2 (c2, x2, y2, s2, c3);
   fulladd stage3 (c3, x3, y3, s3, carryout);

endmodule

module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign s = x ^ y ^ Cin;
   assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

Figure 3.22 Verilog code for a four-bit adder.

3.5.3 USING VECTORED SIGNALS

In Figure 3.22 each of the four-bit inputs and the four-bit output of the adder is represented using single-bit signals. A more convenient approach is to use multibit signals, called *vectors*, to represent the numbers. Just as a number is represented in a logic circuit as signals on multiple wires, it can be represented in Verilog code as a multibit vector. An example of an input vector is

input [3:0] X;

This statement defines X to be a four-bit vector. Its individual bits can be referred to by using an index value in square brackets. Thus, the most-significant bit (MSB) is referred to as $X[3]$ and the least-significant bit (LSB) is $X[0]$. A two-bit vector that consists of the two middle bits of X is denoted as $X[2:1]$. The symbol X refers to the entire vector.

Using vectors we can specify the four-bit adder as shown in Figure 3.23. In addition to the input vectors X and Y, and output vector S, we chose to define the carry signals between the full-adder stages as a three-bit vector $C[3:1]$. Note that the carry into *stage0* is still called *carryin*, while the carry from *stage3* is called *carryout*. The internal carry signals are defined in the statement

wire [3:1] C;

In Figure 3.23, signal C[1] is used to connect the carry output of the full-adder in stage 0 to the carry input of the full-adder in stage 1. Similarly, C[2] and C[3] are used to connect the other stages of the adder.

The vector specification gives the bit width in square brackets, as in $X[3:0]$. The bit width is specified using the index of the MSB first and the LSB last. Hence, $X[3]$ is the MSB and $X[0]$ is the LSB. A reverse ordering can also be used. For example, $Z[0:3]$ defines a four-bit vector in which $Z[0]$ is its MSB and $Z[3]$ is its LSB. The terminology MSB and LSB is natural when vectors are used to represent numbers. In other cases, the bit-select index merely identifies a particular bit in a vector.

```
module adder4 (carryin, X, Y, S, carryout);
   input carryin;
   input [3:0] X, Y;
   output [3:0] S;
   output carryout;
   wire [3:1] C;

   fulladd stage0 (carryin, X[0], Y[0], S[0], C[1]);
   fulladd stage1 (C[1], X[1], Y[1], S[1], C[2]);
   fulladd stage2 (C[2], X[2], Y[2], S[2], C[3]);
   fulladd stage3 (C[3], X[3], Y[3], S[3], carryout);

endmodule
```

Figure 3.23 A four-bit adder using vectors.

3.5.4 USING A GENERIC SPECIFICATION

The approach in designing a ripple-carry adder presented in Figure 3.23 is rather restrictive because the resulting circuit is of a predetermined size of four bits. A similar adder that could add 32-bit numbers would require Verilog code with 32 instances of the full-adder subcircuit defined in separate statements. From the designer's point of view, it is preferable to define a module that could be used to implement an adder of any size, where the size may be given as a parameter.

Verilog allows the use of general parameters that can be given a specific value as desired. For example, an n-bit vector representing a number may be given as $X[n-1:0]$. If n is defined in the Verilog statement

parameter n = 4;

then the bit range of X is [3:0].

The ripple-carry adder in Figure 3.5 can be described using the logic expressions

$$s_k = x_k \oplus y_k \oplus c_k$$
$$c_{k+1} = x_k y_k + x_k c_k + y_k c_k$$

for $k = 0, 1, \ldots, n-1$. Instead of instantiating full-adders as in Figure 3.23, these expressions can be used in Verilog to specify the desired adder.

Figure 3.24 shows Verilog code that defines an n-bit adder. The inputs X and Y and the output sum S are declared to be n-bit vectors. To simplify the use of carry signals in the adder circuit, we defined a vector C that has $n+1$ bits. Bit $C[0]$ is the carry into the LSB position, while $C[n]$ is the carry from the MSB position. Hence $C[0]$ = *carryin* and *carryout* = $C[n]$ in terms of the n-bit adder.

To specify the repetitive structure of the ripple-carry adder, Figure 3.24 introduces the Verilog **for** statement. Like the **if-else** statement introduced in Chapter 2, the **for** statement is a procedural statement that must be placed inside an **always** block, as shown in the figure. As explained in Chapter 2, any signal that is assigned a value by a statement within an **always** block must retain this value until it is again re-evaluated by changes in the sensitivity variables given in the **always** statement. Such signals are declared to be of **reg** type; they are *carryout*, S, and C signals in Figure 3.24. The sensitivity variables are X, Y, and *carryin*.

In our example, the **for** loop consists of two statements delineated by **begin** and **end**. These statements define the sum and carry functions for the adder stage that corresponds to the value of the loop variable k. The range of k is from 0 to $n-1$ and its value is incremented by 1 for each iteration of the loop. The syntax of the Verilog **for** statement is similar to the syntax of a **for** loop in the C programming language. However, the C operators ++ and −− do not exist in Verilog, hence incrementing or decrementing of the loop variable must be given as $k = k + 1$ or $k = k - 1$, rather than k++ or k−−. Note that k is declared to be an integer and it is used to control the number of iterations of the **for** loop; it does not represent a physical connection in the circuit. The effect of the **for** loop is to repeat the statements inside the loop for each loop iteration. For instance, if k were set to 2 in this example, then the **for** loop would be equivalent to the four statements

```
module addern (carryin, X, Y, S, carryout);
   parameter n = 32;
   input carryin;
   input [n– 1:0] X, Y;
   output reg [n –1:0] S;
   output reg carryout;
   reg [n:0] C;
   integer k;

   always @(X, Y, carryin)
   begin
      C[0] = carryin;
      for (k = 0; k < n; k = k+1)
      begin
         S[k] = X[k] ^ Y[k] ^ C[k];
         C[k+1] = (X[k] & Y[k]) | (X[k] & C[k]) | (Y[k] & C[k]);
      end
      carryout = C[n];
   end

endmodule
```

Figure 3.24 A generic specification of a ripple-carry adder.

```
S[0] = X[0] ^ Y[0] ^ C[0];
C[1] = (X[0] & Y[0]) | (X[0] & C[0]) | (Y[0] & C[0]);
S[1] = X[1] ^ Y[1] ^ C[1];
C[2] = (X[1] & Y[1]) | (X[1] & C[1]) | (Y[1] & C[1]);
```

Since the value of *n* is 32, as declared in the **parameter** statement, the code in the figure implements a 32-bit adder.

Using the Generate Capability

In Figure 3.23 we instantiated four copies of the *fulladd* subcircuit to specify a four-bit ripple-carry adder. This approach can be used to specify an *n*-bit adder by using a loop that instantiates the *fulladd* subcircuit *n* times. The Verilog **generate** construct provides the desired capability. It allows instantiation statements to be included inside **for** loops and **if-else** statements. If a **for** loop is included in the **generate** block, the loop index variable has to be declared of type **genvar**. A **genvar** variable is similar to an **integer** variable, but it can have only positive values and it can be used only inside **generate** blocks.

Figure 3.25 shows how the *addern* module can be written to instantiate *n fulladd* modules. Each instance generated by the compiler in the **for** loop will have a unique instance name produced by the compiler based on the **for** loop label. The generated names are *addbit*[0].*stage*, . . . , *addbit*.[$n - 1$].*stage*. This code produces the same result as the code in Figure 3.24.

```
module addern (carryin, X, Y, S, carryout);
    parameter n = 32;
    input carryin;
    input [n–1:0] X, Y;
    output [n–1:0] S;
    output carryout;
    wire [n:0] C;

    genvar i;
    assign C[0] = carryin;
    assign carryout = C[n];

    generate
        for (i = 0; i <= n–1; i = i+1)
        begin:addbit
            fulladd stage (C[i], X[i], Y[i], S[i], C[i+1]);
        end
    endgenerate

endmodule

module fulladd (Cin, x, y, s, Cout);
    input Cin, x, y;
    output s, Cout;

    assign s = x ^ y ^ Cin;
    assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

Figure 3.25 A ripple-carry adder specified by using the **generate** statement.

3.5.5 NETS AND VARIABLES IN VERILOG

A logic circuit is modeled in Verilog by a collection of interconnected logic elements and/or by procedural statements that describe its behavior. Connections between logic elements are defined using *nets*. Signals produced by procedural statements are referred to as *variables*.

Nets

A net represents a node in a circuit. Nets can be of different types. For synthesis purposes the only important nets are of **wire** type, which we used in Section 3.5.3. A **wire** connects an output of one logic element in a circuit to an input of another logic element.

It can be a *scalar* that represents a single connection or a *vector* that represents multiple connections. For example, in Figure 3.22, carry signals c_3, c_2, and c_1 are scalars that model the connections between the full-adder modules. The specific connections are defined by the way the full-adder modules are instantiated. In Figure 3.23, the same carry signals are defined as a three-bit vector C. Observe that in Figure 3.22 the carry signals are not explicitly declared to be of **wire** type. The reason is that nets do not have to be declared in the code because Verilog syntax assumes that all signals are nets by default. Of course, the code in the figure would also be correct if we include in it the declaration

```
wire c3, c2, c1;
```

In Figure 3.23 it is necessary to declare the existence of vector C; otherwise, the Verilog compiler would not be able to determine that $C[3]$, $C[2]$, and $C[1]$ are the constituent signals of C. Since these signals are nets, the vector C is declared to be of **wire** type.

Variables

Verilog provides *variables* to allow a circuit to be described in terms of its behavior. A variable can be assigned a value in one Verilog statement, and it retains this value until it is overwritten by a subsequent assignment statement. There are two types of variables: **reg** and **integer**. As mentioned in Chapter 2, all signals that are assigned a value using procedural statements must be declared as variables by using the **reg** or **integer** keywords. The scalar *carryout* and the vectors S and C in Figure 3.24 are examples of the **reg** type. The loop variable k in the same figure illustrates the **integer** type. It serves as a loop index. Such variables are useful for describing a circuit's behavior; they do not usually correspond directly to signals in the resulting circuit.

Further discussion of nets and variables is given in Appendix A.

3.5.6 ARITHMETIC ASSIGNMENT STATEMENTS

Arithmetic operations are used so often that it is convenient to have them incorporated directly into a hardware description language. Verilog implements such operations using arithmetic assignment statements and vectors. If the following vectors are defined

```
input  [n−1:0] X, Y;
output [n−1:0] S;
```

then, the arithmetic assignment statement

```
S = X + Y;
```

represents an n-bit adder.

In addition to the + operator, which is used for addition, Verilog also provides other arithmetic operators. The Verilog operators are discussed fully in Chapter 4 and Appendix A. The complete Verilog code that includes the preceding statement is given in Figure 3.26. Since there is a single statement in the **always** block, it is not necessary to include the **begin** and **end** delimiters.

The code in Figure 3.26 defines a circuit that generates the n sum bits, but it does not include carry-out or overflow signals. As explained previously, the carry-out signal is useful when the numbers X, Y, and S are interpreted as being unsigned numbers. The carry-out is 1 when the sum of unsigned numbers overflows n bits. But, if X, Y, and S are interpreted as being signed numbers, then the carry-out is not meaningful, and we have to generate the arithmetic overflow output as discussed in Section 3.3.5. One way in which these signals can be generated is given in Figure 3.27.

The carry-out from the MSB position, $n-1$, can be derived by observing that the carry-out must be 1 if both x_{n-1} and y_{n-1} are 1, or if either x_{n-1} or y_{n-1} is 1 and s_{n-1} is 0.

```
module addern (carryin, X, Y, S);
   parameter n = 32;
   input carryin;
   input [n–1:0] X, Y;
   output reg [n–1:0] S;

   always @(X, Y, carryin)
      S = X + Y + carryin;

endmodule
```

Figure 3.26 Specification of an n-bit adder using arithmetic assignment.

```
module addern (carryin, X, Y, S, carryout, overflow);
   parameter n = 32;
   input carryin;
   input [n– 1:0] X, Y;
   output reg [n– 1:0] S;
   output reg carryout, overflow;

   always @(X, Y, carryin)
   begin
      S = X + Y + carryin;
      carryout = (X[n– 1] & Y[n– 1]) | (X[n– 1] &~ S[n– 1]) | (Y[n– 1] &~ S[n– 1]);
      overflow = (X[n– 1] & Y[n– 1] &~ S[n– 1]) | (~X[n– 1] & ~Y[n– 1] & S[n– 1]);
   end

endmodule
```

Figure 3.27 An n-bit adder with carry-out and overflow signals.

Thus,

$$carryout = x_{n-1}y_{n-1} + x_{n-1}\overline{s}_{n-1} + y_{n-1}\overline{s}_{n-1}$$

(Note that this is just a normal logic expression in which the + sign represents the OR operation.) The expression for arithmetic overflow was defined in Section 3.3.5 as $c_n \oplus c_{n-1}$. In our case, c_n corresponds to *carryout*, but there is no direct way of accessing c_{n-1}, which is the carry from bit-position $n-2$. Instead, we can use the more intuitive expression derived in Section 3.3.5, so that

$$overflow = x_{n-1}y_{n-1}\overline{s}_{n-1} + \overline{x}_{n-1}\overline{y}_{n-1}s_{n-1}$$

Another way of including the carry-out and overflow signals is shown in Figure 3.28. The $(n+1)$-bit vector named *Sum* is used. The extra bit, *Sum*[*n*], becomes the carry-out from bit-position $n-1$ in the adder. The statement used to assign the sum of *X*, *Y*, and *carryin* to the *Sum* signal uses an unusual syntax. The meaning of the terms in brackets, namely {1'b0, X} and {1'b0, Y}, is that a 0 is concatenated on the left of the *n*-bit vectors *X* and *Y* to create (*n*+1)-bit vectors. In Verilog the { , } operator is called the *concatenate* operator. If *A* is an *m*-bit vector and *B* is a *k*-bit vector, then {A, B} creates an $(m+k)$-bit vector comprising *A* as its most-significant *m* bits and *B* as its least-significant *k* bits. The notation 1'b0 represents a one-bit binary number that has the value 0. The reason that the concatenate operator is used in Figure 3.28 is to cause *Sum*[*n*] to be equivalent to the carry from bit position $n-1$. In effect, we created $x_n = y_n = 0$ so that

$$\text{Sum}[n] = 0 + 0 + c_{n-1}$$

```
module addern (carryin, X, Y, S, carryout, overflow);
   parameter n = 32;
   input carryin;
   input [n-1:0] X, Y;
   output reg [n-1:0] S;
   output reg carryout, overflow;
   reg [n:0] Sum;

   always @(X, Y, carryin)
   begin
      Sum = {1'b0, X} + {1'b0, Y} + carryin;
      S = Sum[n-1:0];
      carryout = Sum[n];
      overflow = (X[n-1] & Y[n-1] & ~S[n-1]) | (~X[n-1] & ~Y[n-1] & S[n-1]);
   end

endmodule
```

Figure 3.28 An alternative specification of *n*-bit adder with carry-out and overflow signals.

This example is useful because it provides a simple introduction to the concept of concatenation. But we could have written simply

$$\text{Sum} = \text{X} + \text{Y} + \text{carryin};$$

Because *Sum* is an $(n + 1)$-bit vector, the summation will be performed as if X and Y were $(n + 1)$-bit vectors in which 0s are padded on the left.

Another detail to observe from the figure is the statement

$$\text{S} = \text{Sum[n}-\text{1:0]};$$

This statement assigns the lower n bits of *Sum* to the output sum S. The next statement assigns the carry-out from the addition, *Sum*[n], to the output signal *carryout*.

We show the code in Figures 3.27 and 3.28 to illustrate some features of Verilog in the context of adder design. In general, a given design task can be performed using different approaches, as we will see throughout the book. Let us attempt another specification of the n-bit adder. In Figure 3.28 we use an $(n + 1)$-bit vector, *Sum*, as an intermediate signal needed to produce the n bits of S and the carry-out from the adder stage $n - 1$. This requires two Verilog statements that extract the desired bits from *Sum*. We showed how concatenation can be used to pad a 0 to vectors X and Y, but pointed out that this is not necessary because a vector is automatically padded with 0s if it is involved in an arithmetic operation that produces a result of greater bit size. We can use concatenation more effectively on the left side of the addition statement by concatenating *carryout* to the S vector so that

$$\{\text{carryout, S}\} = \text{X} + \text{Y} + \text{carryin};$$

Then there is no need for the *Sum* signal and the Verilog code is simplified as indicated in Figure 3.29. Since both figures, 3.28 and 3.29, describe the same behavior of the adder, the Verilog compiler is likely to generate the same circuit for either figure. The code in Figure 3.29 is simpler and more elegant.

```
module addern (carryin, X, Y, S, carryout, overflow);
   parameter n = 32;
   input carryin;
   input [n-1:0] X, Y;
   output reg [n-1:0] S;
   output reg carryout, overflow;

   always @(X, Y, carryin)
   begin
      {carryout, S} = X + Y + carryin;
      overflow = (X[n-1] & Y[n-1] & ~S[n-1]) | (~X[n-1] & ~Y[n-1] & S[n-1]);
   end

endmodule
```

Figure 3.29 Simplified complete specification of n-bit adder.

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output reg s, Cout;

   always @(x, y, Cin)
      {Cout, s} = x + y + Cin;

endmodule
```

Figure 3.30 Behavioral specification of a full-adder.

Note that the same approach can be used to specify a full-adder circuit, as shown in Figure 3.30. Unlike the specifications in Figures 3.18 to 3.21, which define the structure of the full-adder in terms of basic logic operations, in this case the code describes its behavior and the Verilog compiler implements the suitable details using the target technology.

3.5.7 MODULE HIERARCHY IN VERILOG CODE

In Section 2.10.3 we introduced hierarchical Verilog code, in which a Verilog module instantiates other Verilog modules as subcircuits. If an instantiated module includes parameters, then either the default value of a parameter can be used for each instance, or a new value of the parameter can be specified.

Suppose that we wish to create a circuit consisting of two adders as follows. One adder generates a 16-bit sum $S = A + B$, while the other generates an 8-bit sum $T = C + D$. An overflow signal must be set to 1 if either of the adders generates an arithmetic overflow. One way of specifying the desired circuit is presented in Figure 3.31. The top-level module, *adder_hier*, instantiates two instances of the *addern* module in Figure 3.29. Since the default value of n is 32 for *addern*, the top-level module has to define the desired values of 16 and 8. The statement

```
defparam U1.n = 16
```

sets the value of n to 16 for the *addern* instance called $U1$. Similarly, the statement

```
defparam U2.n = 8
```

sets the value of n to 8 for the instance named $U2$.

Note that in Figure 3.31 we added an extra bit to both S and T to hold the carry-out signals produced by the adders; these signals are useful if the resulting sums are interpreted as unsigned numbers.

Another way in which the value of n can be specified in the instantiated modules is shown in Figure 3.32. Here, the value of n is defined by using the Verilog # operator, instead of using the **defparam** statement. The code in Figures 3.31 and 3.32 produces identical results.

```
module adder_hier (A, B, C, D, S, T, overflow);
   input [15:0] A, B;
   input [7:0] C, D;
   output [16:0] S;
   output [8:0] T;
   output overflow;

   wire o1, o2; // used for the overflow signals

   addern U1 (1'b0, A, B, S[15:0], S[16], o1);
   defparam U1.n = 16;
   addern U2 (1'b0, C, D, T[7:0], T[8], o2);
   defparam U2.n = 8;

   assign overflow = o1 | o2;

endmodule
```

Figure 3.31 An example of setting parameter values in Verilog code.

```
module adder_hier (A, B, C, D, S, T, overflow);
   input [15:0] A, B;
   input [7:0] C, D;
   output [16:0] S;
   output [8:0] T;
   output overflow;

   wire o1, o2; // used for the overflow signals

   addern #(16) U1 (1'b0, A, B, S[15:0], S[16], o1);
   addern #(8) U2 (1'b0, C, D, T[7:0], T[8], o2);

   assign overflow = o1 | o2;

endmodule
```

Figure 3.32 Using the Verilog # operator to set the values of parameters.

An alternative version of the code in Figure 3.32 is given in Figure 3.33. Here, we have again used the # operator, but in this case we have explicitly included the name *n* by using the syntax #(.n(16)) and #(.n(8)). In the same way that the names of parameters can be shown explicitly, the code in Figure 3.33 illustrates how the port names of a subcircuit

```
module adder_hier (A, B, C, D, S, T, overflow);
    input [15:0] A, B;
    input [7:0] C, D;
    output [16:0] S;
    output [8:0] T;
    output overflow;

    wire o1, o2; // used for the overflow signals

    addern #(.n(16)) U1
    (
        .carryin (1'b0),
        .X (A),
        .Y (B),
        .S (S[15:0]),
        .carryout (S[16]),
        .overflow (o1)
    );
    addern #(.n(8)) U2
    (
        .carryin (1'b0),
        .X (C),
        .Y (D),
        .S (T[7:0]),
        .carryout (T[8]),
        .overflow (o2)
    );

    assign overflow = o1 | o2;

endmodule
```

Figure 3.33 An alternative version of the code in Figure 3.32.

can be included explicitly. In Verilog jargon this style of code is referred to as *named* port connections. In this case, the signals can be listed in any order. If the names of port signals of the instantiated module are not shown explicitly, as in Figures 3.31 and 3.32, then the order in which signals are listed in an instantiation statement determines how the connections are made. This is referred to as *ordered* association of port connections. In this case, the instantiation statement must list the ports in the order in which they appear in the instantiated module. The drawback of the style of code in Figure 3.33 is that it is more verbose. But, it has the advantage of being more explicit and hence less susceptible to careless errors.

3.5.8 REPRESENTATION OF NUMBERS IN VERILOG CODE

Numbers can be given as constants in Verilog code. They can be given as binary (b), octal (o), hexadecimal (h), or decimal (d) numbers. Their size can be either fixed or unspecified. For sized numbers the format is

<size_in_bits>'<radix_identifier><significant_digits>

The size is a decimal number that gives the number of bits needed, the radix is identified using letters b, o, h, or d, and the digits are given in the notation of the radix used. For example, the decimal number 2217 can be represented using 12 bits as follows

```
12'b100010101001
12'o4251
12'h8A9
12'd2217
```

Unsized numbers are given without specifying the size. For example, the decimal number 278 may be given as

```
'b100010110
'o426
'h116
278
```

For decimal numbers it is not necessary to give the radix identifier d. When an unsized number is used in an expression the Verilog compiler gives it a certain size, which is typically the same as the size of the other operand(s) in the expression. To improve readability, it is possible to use the underscore character. Instead of writing 12'b100010101001, it is easier to visualize the same number as 12'b1000_1010_1001.

Negative numbers are represented by placing the minus sign in front. Thus, if −5 is specified as −4'b101, it will be interpreted as a four-bit 2's-complement of 5, which is 1011.

The specified size may exceed the number of bits that are actually needed to represent a given number. In this case, the final representation is padded to the left to yield the required size. However, if there are more digits than can fit into the number of bits given as the size, the extra digits will be ignored.

Numbers represented by vectors of different bit sizes can be used in arithmetic operations. Suppose that *A* is an eight-bit vector and *B* is a four-bit vector. Then the statement

```
S = A + B;
```

will generate an eight-bit sum vector *S*. The result will be correct if *B* is a positive number. However, if *B* is a negative number expressed in 2's complement representation, the result will be incorrect because 0s will be padded on the left to make *B* an eight-bit vector for the purpose of the addition operation. The value of a positive number does not change if 0s are

appended as the most-significant bits; the value of a negative number does not change if 1s are appended as the most-significant bits. Such replication of the sign bit is called *sign extension*. Therefore, for correct operation it is necessary to use a sign-extended version of B, which can be accomplished with concatenation in the statement

$$\text{S} = \text{A} + \{4\{\text{B}[3]\}, \text{B}\};$$

The notation 4{B[3]} denotes that the bit B[3] is replicated four times; it is equivalent to writing {B[3], B[3], B[3], B[3]}. This is referred to as the *replication* operator, which is discussed in Chapter 4.

3.6 MULTIPLICATION

Before we discuss the general issue of multiplication, we should note that a binary number, B, can be multiplied by 2 simply by adding a zero to the right of its least-significant bit. This effectively moves all bits of B to the left, and we say that B is *shifted* left by one bit position. Thus if $B = b_{n-1}b_{n-2}\cdots b_1b_0$, then $2 \times B = b_{n-1}b_{n-2}\cdots b_1b_0 0$. (We have already used this fact in Section 3.2.3.) Similarly, a number is multiplied by 2^k by shifting it left by k bit positions. This is true for both unsigned and signed numbers.

We should also consider what happens if a binary number is shifted right by k bit positions. According to the positional number representation, this action divides the number by 2^k. For unsigned numbers the shifting amounts to adding k zeros to the left of the most-significant bit. For example, if B is an unsigned number, then $B \div 2 = 0b_{n-1}b_{n-2}\cdots b_2b_1$. Note that bit b_0 is lost when shifting to the right. For signed numbers it is necessary to preserve the sign. This is done by shifting the bits to the right and filling from the left with the value of the sign bit. Hence if B is a signed number, then $B \div 2 = b_{n-1}b_{n-1}b_{n-2}\cdots b_2b_1$. For instance, if $B = 011000 = (24)_{10}$, then $B \div 2 = 001100 = (12)_{10}$ and $B \div 4 = 000110 = (6)_{10}$. Similarly, if $B = 101000 = -(24)_{10}$, then $B \div 2 = 110100 = -(12)_{10}$ and $B \div 4 = 111010 = -(6)_{10}$. The reader should also observe that the smaller the positive number, the more 0s there are to the left of the first 1, while for a negative number there are more 1s to the left of the first 0.

Now we can turn our attention to the general task of multiplication. Two binary numbers can be multiplied using the same method as we use for decimal numbers. We will focus our discussion on multiplication of unsigned numbers. Figure 3.34*a* shows how multiplication is performed manually, using four-bit numbers. Each multiplier bit is examined from right to left. If a bit is equal to 1, an appropriately shifted version of the multiplicand is added to form a *partial product*. If the multiplier bit is equal to 0, then nothing is added. The sum of all shifted versions of the multiplicand is the desired product. Note that the product occupies eight bits.

3.6.1 ARRAY MULTIPLIER FOR UNSIGNED NUMBERS

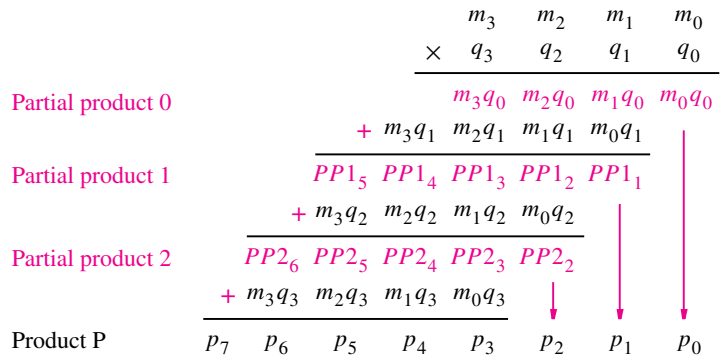
Figure 3.34*b* indicates how multiplication may be performed by using multiple adders. In each step a four-bit adder is used to compute the new partial product. Note that as the computation progresses, the least-significant bits are not affected by subsequent additions;

Multiplicand M	(14)	1 1 1 0
Multiplier Q	(11)	× 1 0 1 1
		1 1 1 0
		1 1 1 0
		0 0 0 0
		1 1 1 0
Product P	(154)	1 0 0 1 1 0 1 0

(a) Multiplication by hand

Multiplicand M	(14)	1 1 1 0
Multiplier Q	(11)	× 1 0 1 1
Partial product 0		1 1 1 0
		+ 1 1 1 0
Partial product 1		1 0 1 0 1
		+ 0 0 0 0
Partial product 2		0 1 0 1 0
		+ 1 1 1 0
Product P	(154)	1 0 0 1 1 0 1 0

(b) Using multiple adders



(c) Hardware implementation

Figure 3.34 Multiplication of unsigned numbers.

hence they can be passed directly to the final product, as indicated by blue arrows. Of course, these bits are a part of the partial products as well.

The same scheme can be used to design a multiplier circuit. We will stay with four-bit numbers to keep the discussion simple. Let the multiplicand, multiplier, and product be denoted as $M = m_3m_2m_1m_0$, $Q = q_3q_2q_1q_0$, and $P = p_7p_6p_5p_4p_3p_2p_1p_0$, respectively. Figure 3.34*c* shows the required operations. Partial product 0 is obtained by using the AND of q_0 with each bit of M, which produces 0 if $q_0 = 0$ and M if $q_0 = 1$. Thus

$$PP0 = m_3q_0 \;\; m_2q_0 \;\; m_1q_0 \;\; m_0q_0$$

Partial product 1, $PP1$, is generated by adding $PP0$ to a shifted version of M that is ANDed with q_1. Similarly, partial product $PP2$ is generated using the AND of q_2 with M shifted by another bit position and adding to $PP1$, and so on.

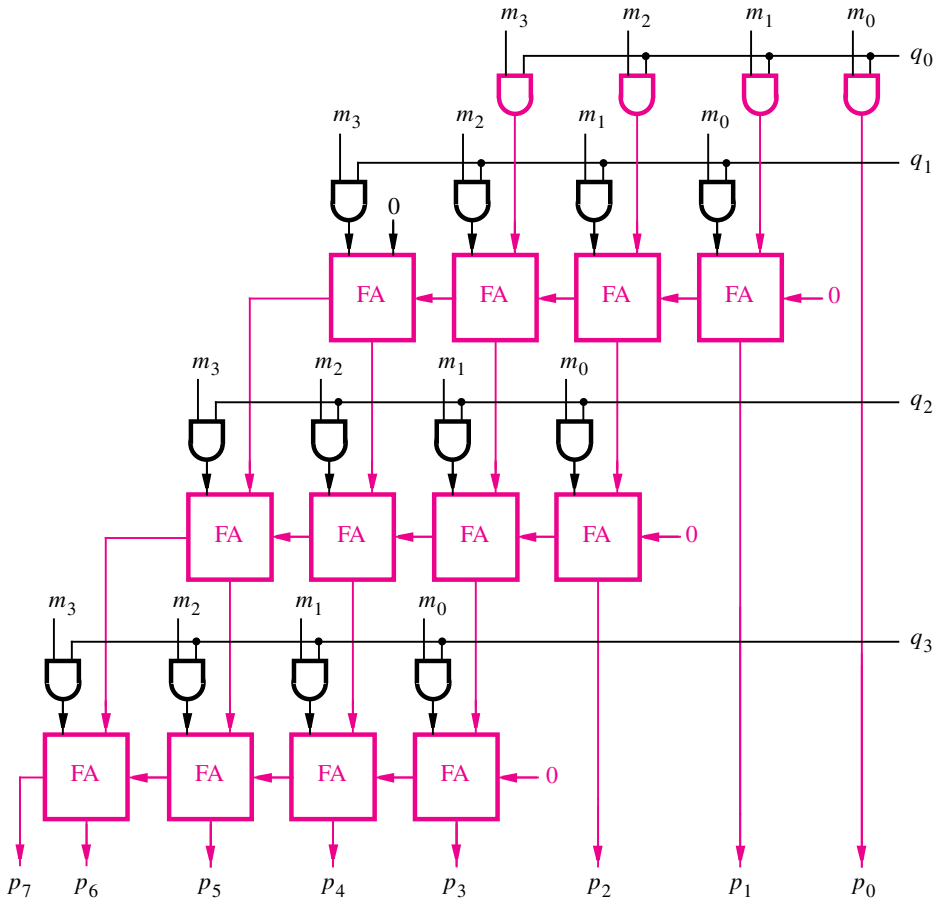


Figure 3.35 A 4 × 4 multiplier circuit.

A circuit that implements the preceding operations is arranged in an array, as shown in Figure 3.35. In this figure the AND gates and full-adders that produce the partial products are shown in blue, in the same way as the blue highlighted rows in Figure 3.34*c*. The full-adders are connected to form ripple-carry adders. It is possible to design even faster multipliers by using other types of adders [1].

3.6.2 MULTIPLICATION OF SIGNED NUMBERS

Multiplication of unsigned numbers illustrates the main issues involved in the design of multiplier circuits. Multiplication of signed numbers is somewhat more complex.

If the multiplier operand is positive, it is possible to use essentially the same scheme as for unsigned numbers. For each bit of the multiplier operand that is equal to 1, a properly

shifted version of the multiplicand must be added to the partial product. The multiplicand can be either positive or negative.

Since shifted versions of the multiplicand are added to the partial products, it is important to ensure that the numbers involved are represented correctly. For example, if the two right-most bits of the multiplier are both equal to 1, then the first addition must produce the partial product $PP1 = M + 2M$, where M is the multiplicand. If $M = m_{n-1}m_{n-2}\cdots m_1m_0$, then $PP1 = m_{n-1}m_{n-2}\cdots m_1m_0 + m_{n-1}m_{n-2}\cdots m_1m_00$. The adder that performs this addition comprises circuitry that adds two operands of equal length. Since shifting the multiplicand to the left, to generate $2M$, results in one of the operands having $n + 1$ bits, the required addition has to be performed using the second operand, M, represented also as an $(n + 1)$-bit number. An n-bit signed number is represented as an $(n + 1)$-bit number by using sign extension, that is, by replicating the sign bit as the new left-most bit. Thus $M = m_{n-1}m_{n-2}\cdots m_1m_0$ is represented using $(n + 1)$ bits as $M = m_{n-1}m_{n-1}m_{n-2}\cdots m_1m_0$.

When a shifted version of the multiplicand is added to a partial product, overflow has to be avoided. Hence the new partial product must be larger by one extra bit. Figure 3.36*a* illustrates the process of multiplying two positive numbers. The sign-extended bits are shown in blue. Part (*b*) of the figure involves a negative multiplicand. Note that the resulting product has $2n$ bits in both cases.

For a negative multiplier operand, it is possible to convert both the multiplier and the multiplicand into their 2's complements because this will not change the value of the result. Then the scheme for a positive multiplier can be used.

We have presented a relatively simple scheme for multiplication of signed numbers. There exist other techniques that are more efficient but also more complex. We will not pursue these techniques, but an interested reader may consult reference [1].

We have discussed circuits that perform addition, subtraction, and multiplication. Another arithmetic operation that is needed in computer systems is division. Circuits that perform division are more complex; we will present an example in Chapter 7. Various techniques for performing division are usually discussed in books on the subject of computer organization and can be found in references [1, 2].

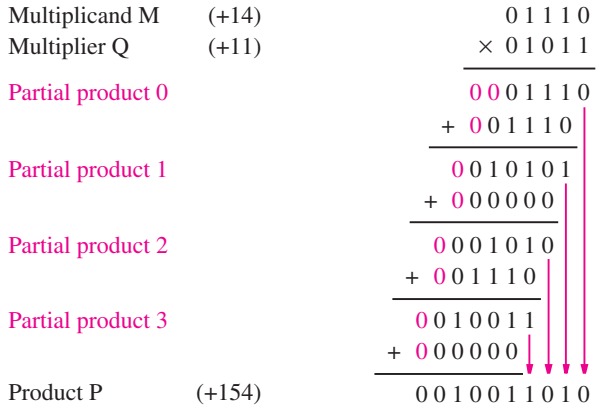
3.7 OTHER NUMBER REPRESENTATIONS

In the previous sections we dealt with binary integers represented in the positional number representation. Other types of numbers are also used in digital systems. In this section we will discuss briefly three other types: fixed-point, floating-point, and binary-coded decimal numbers.

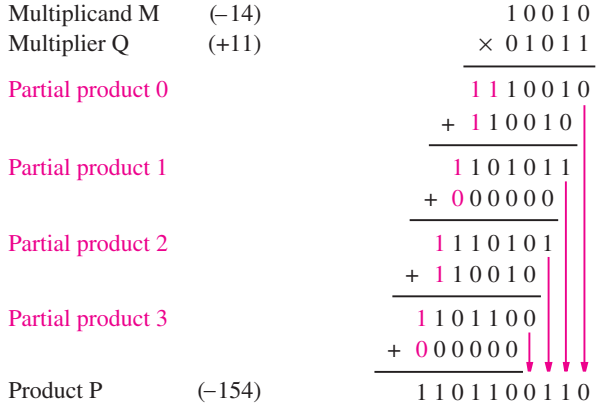
3.7.1 FIXED-POINT NUMBERS

A *fixed-point* number consists of integer and fraction parts. It can be written in the positional number representation as

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



(a) Positive multiplicand



(b) Negative multiplicand

Figure 3.36 Multiplication of signed numbers.

The value of the number is

$$V(B) = \sum_{i=-k}^{n-1} b_i \times 2^i$$

The position of the radix point is assumed to be fixed; hence the name fixed-point number. If the radix point is not shown, then it is assumed to be to the right of the least-significant digit, which means that the number is an integer.

Logic circuits that deal with fixed-point numbers are essentially the same as those used for integers. We will not discuss them separately.

3.7.2 FLOATING-POINT NUMBERS

Fixed-point numbers have a range that is limited by the significant digits used to represent the number. For example, if we use eight digits and a sign to represent decimal integers, then the range of values that can be represented is 0 to ±99999999. If eight digits are used to represent a fraction, then the representable range is 0.00000001 to ±0.99999999. In scientific applications it is often necessary to deal with numbers that are very large or very small. Instead of using the fixed-point representation, which would require many significant digits, it is better to use the floating-point representation in which numbers are represented by a *mantissa* comprising the significant digits and an *exponent* of the radix *R*. The format is

$$Mantissa \times R^{Exponent}$$

The numbers are often *normalized*, such that the radix point is placed to the right of the first nonzero digit, as in 5.234×10^{43} or 6.31×10^{-28}.

Binary floating-point representation has been standardized by the Institute of Electrical and Electronic Engineers (IEEE) [3]. Two sizes of formats are specified in this standard—a *single-precision* 32-bit format and a *double-precision* 64-bit format. Both formats are illustrated in Figure 3.37.

Single-Precision Floating-Point Format

Figure 3.37*a* depicts the single-precision format. The left-most bit is the sign bit—0 for positive and 1 for negative numbers. There is an 8-bit exponent field, *E*, and a 23-bit mantissa field, *M*. The exponent is with respect to the radix 2. Because it is necessary to

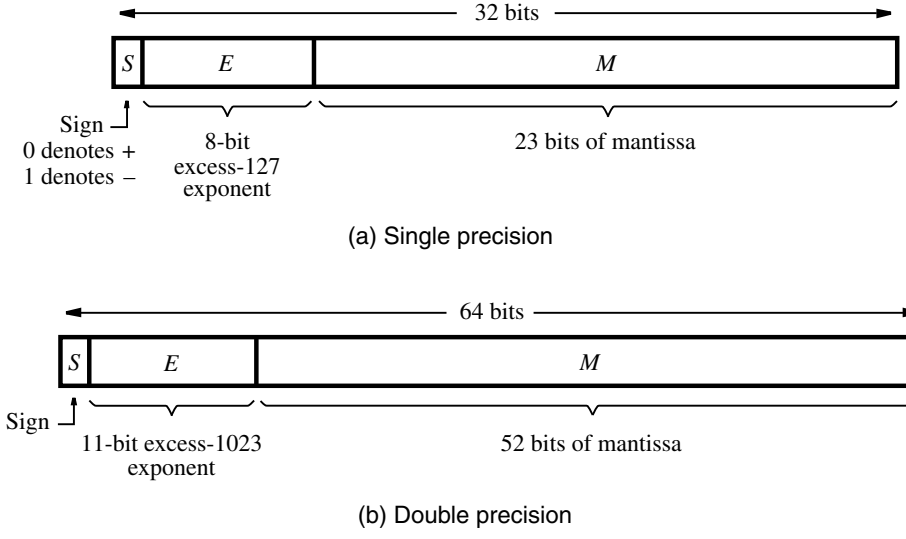


Figure 3.37 IEEE Standard floating-point formats.

be able to represent both very large and very small numbers, the exponent can be either positive or negative. Instead of simply using an 8-bit signed number as the exponent, which would allow exponent values in the range −128 to 127, the IEEE standard specifies the exponent in the *excess-127* format. In this format the value 127 is added to the value of the actual exponent so that

$$Exponent = E - 127$$

In this way E becomes a positive integer. This format is convenient for adding and subtracting floating-point numbers because the first step in these operations involves comparing the exponents to determine whether the mantissas must be appropriately shifted to add/subtract the significant bits. The range of E is 0 to 255. The extreme values of $E = 0$ and $E = 255$ are taken to denote the exact zero and infinity, respectively. Therefore, the normal range of the exponent is −126 to 127, which is represented by the values of E from 1 to 254.

The mantissa is represented using 23 bits. The IEEE standard calls for a normalized mantissa, which means that the most-significant bit is always equal to 1. Thus it is not necessary to include this bit explicitly in the mantissa field. Therefore, if M is the bit vector in the mantissa field, the actual value of the mantissa is $1.M$, which gives a 24-bit mantissa. Consequently, the floating-point format in Figure 3.37*a* represents the number

$$\text{Value} = \pm 1.M \times 2^{E-127}$$

The size of the mantissa field allows the representation of numbers that have the precision of about seven decimal digits. The exponent field range of 2^{-126} to 2^{127} corresponds to about $10^{\pm 38}$.

Double-Precision Floating-Point Format

Figure 3.37*b* shows the double-precision format, which uses 64 bits. Both the exponent and mantissa fields are larger. This format allows greater range and precision of numbers. The exponent field has 11 bits, and it specifies the exponent in the *excess-1023* format, where

$$Exponent = E - 1023$$

The range of E is 0 to 2047, but again the values $E = 0$ and $E = 2047$ are used to indicate the exact zero and infinity, respectively. Thus the normal range of the exponent is −1022 to 1023, which is represented by the values of E from 1 to 2046.

The mantissa field has 52 bits. Since the mantissa is assumed to be normalized, its actual value is again $1.M$. Therefore, the value of a floating-point number is

$$\text{Value} = \pm 1.M \times 2^{E-1023}$$

This format allows representation of numbers that have the precision of about 16 decimal digits and the range of approximately $10^{\pm 308}$.

Arithmetic operations using floating-point operands are significantly more complex than signed integer operations. Because this is a rather specialized domain, we will not elaborate on the design of logic circuits that can perform such operations. For a more complete discussion of floating-point operations, the reader may consult references [1, 2].

3.7.3 BINARY-CODED-DECIMAL REPRESENTATION

In digital systems it is possible to represent decimal numbers simply by encoding each digit in binary form. This is called the *binary-coded-decimal (BCD)* representation. Because there are 10 digits to encode, it is necessary to use four bits per digit. Each digit is encoded by the binary pattern that represents its unsigned value, as shown in Table 3.3. Note that only 10 of the 16 available patterns are used in BCD, which means that the remaining 6 patterns should not occur in logic circuits that operate on BCD operands; these patterns are usually treated as don't-care conditions in the design process. BCD representation was used in some early computers as well as in many handheld calculators. Its main virtue is that it provides a format that is convenient when numerical information is to be displayed on a simple digit-oriented display. Its drawbacks are complexity of circuits that perform arithmetic operations and the fact that six of the possible code patterns are wasted.

Even though the importance of BCD representation has diminished, it is still encountered. To give the reader an indication of the complexity of the required circuits, we will consider BCD addition in some detail.

BCD Addition

The addition of two BCD digits is complicated by the fact that the sum may exceed 9, in which case a correction will have to be made. Let $X = x_3x_2x_1x_0$ and $Y = y_3y_2y_1y_0$ represent the two BCD digits and let $S = s_3s_2s_1s_0$ be the desired sum digit, $S = X + Y$. Obviously, if $X + Y \leq 9$, then the addition is the same as the addition of 2 four-bit unsigned binary numbers. But, if $X + Y > 9$, then the result requires two BCD digits. Moreover, the four-bit sum obtained from the four-bit adder may be incorrect.

There are two cases where some correction has to be made: when the sum is greater than 9 but no carry-out is generated using four bits, and when the sum is greater than 15 so

Table 3.3 Binary-coded decimal digits.

Decimal digit	BCD code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

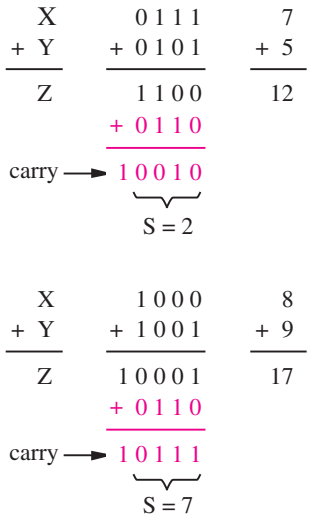


Figure 3.38 Addition of BCD digits.

that a carry-out is generated using four bits. Figure 3.38 illustrates these cases. In the first case the four-bit addition yields $Z = 7 + 5 = 12$. To obtain a correct BCD result, we must generate $S = 2$ and a carry-out of 1. The necessary correction is apparent from the fact that the four-bit addition is a modulo-16 scheme, whereas decimal addition is a modulo-10 scheme. Therefore, a correct decimal digit can be generated by adding 6 to the result of four-bit addition whenever this result exceeds 9. Thus we can arrange the computation as follows

$$Z = X + Y$$
$$\text{If } Z \leq 9, \text{ then } S = Z \text{ and carry-out} = 0$$
$$\text{if } Z > 9, \text{ then } S = Z + 6 \text{ and carry-out} = 1$$

The second example in Figure 3.38 shows what happens when $X + Y > 15$. In this case $Z = 17$, which is a five-bit binary number. Note that the four least-significant bits, Z_{3-0}, represent the value 1, while Z_4 represents the value 16. To generate a correct BCD result, it is again necessary to add 6 to the intermediate sum Z_{3-0} to produce the value 7. The bit Z_4 is the carry to the next digit position.

Figure 3.39 gives a block diagram of a one-digit BCD adder that is based on this scheme. The block that detects whether $Z > 9$ produces an output signal, Adjust, which controls the multiplexer that provides the correction when needed. A second four-bit adder generates the corrected sum bits. Since whenever the adjustment of 6 is made there is a carry to the next digit position, c_{out} is just equal to the Adjust signal.

The one-digit BCD adder can be specified in Verilog code by describing its behavior as shown in Figure 3.40. Inputs X and Y, and output S are defined as four-bit numbers. The intermediate sum, Z, is defined as a five-bit number. The **if-else** statement is used to

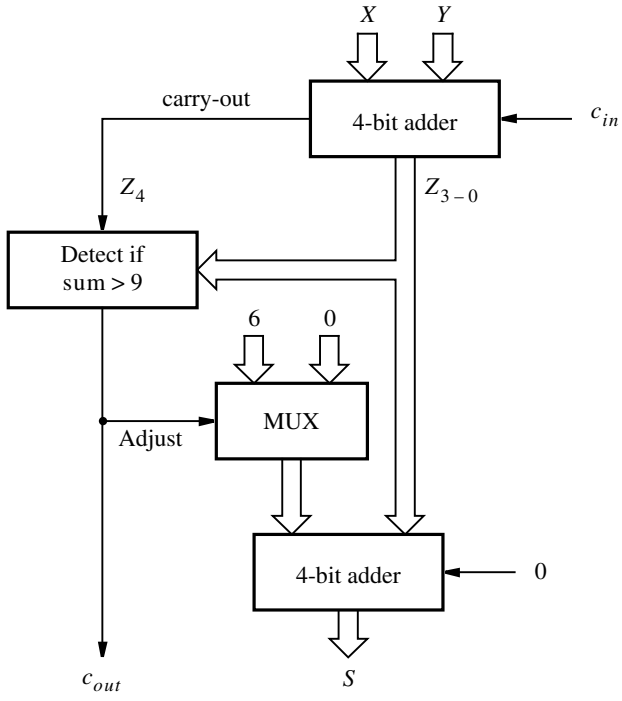


Figure 3.39 Block diagram for a one-digit BCD adder.

```
module bcdadd (Cin, X, Y, S, Cout);
   input Cin;
   input [3:0] X, Y;
   output reg [3:0] S;
   output reg Cout;
   reg [4:0] Z;

   always @(X, Y, Cin)
   begin
      Z = X + Y + Cin;
      if (Z < 10)
         {Cout, S} = Z;
      else
         {Cout, S} = Z + 6;
   end

endmodule
```

Figure 3.40 Verilog code for a one-digit BCD adder.

provide the adjustment explained above; hence it is not necessary to use an explicit *Adjust* signal.

If we wish to derive a circuit to implement the block diagram in Figure 3.39 by hand, instead of by using Verilog, then the following approach can be used. To define the *Adjust* function, we can observe that the intermediate sum will exceed 9 if the carry-out from the four-bit adder is equal to 1, or if $z_3 = 1$ and either z_2 or z_1 (or both) are equal to 1. Hence the logic expression for this function is

$$\text{Adjust} = \text{Carry-out} + z_3(z_2 + z_1)$$

Instead of implementing another complete four-bit adder to perform the correction, we can use a simpler circuit because the addition of constant 6 does not require the full capability of a four-bit adder. Note that the least-significant bit of the sum, s_0, is not affected at all; hence $s_0 = z_0$. A two-bit adder may be used to develop bits s_2 and s_1. Bit s_3 is the same as z_3 if the carry-out from the two-bit adder is 0, and it is equal to $\overline{z}_3$ if this carry-out is equal to 1. A complete circuit that implements this scheme is shown in Figure 3.41. Using the

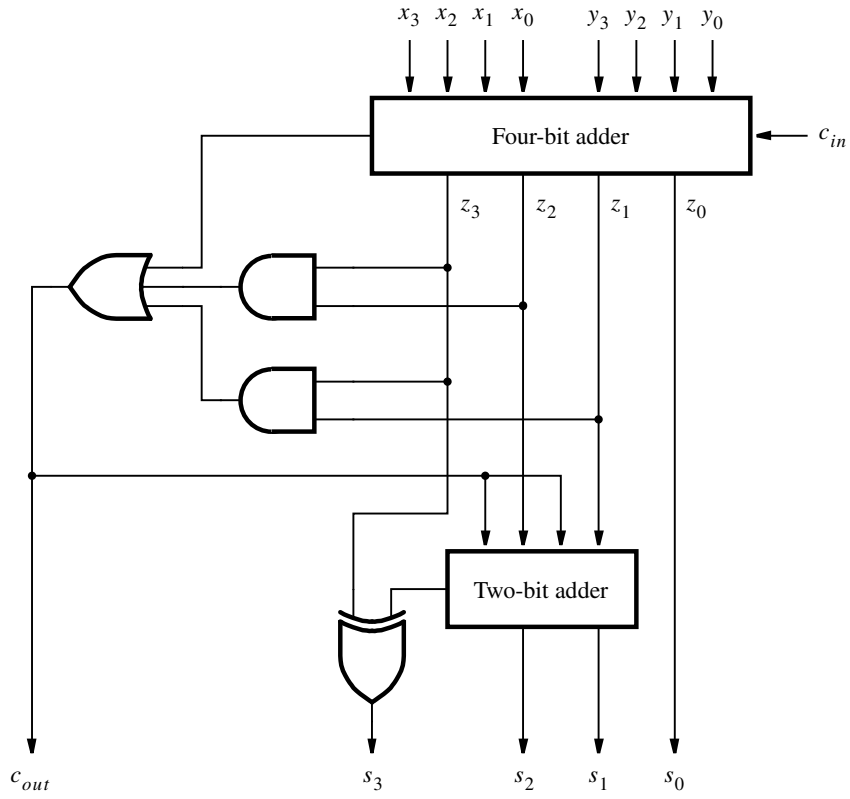


Figure 3.41 Circuit for a one-digit BCD adder.

one-digit BCD adder as a basic block, it is possible to build larger BCD adders in the same way as a binary full-adder is used to build larger ripple-carry binary adders.

Subtraction of BCD numbers can be handled with the radix-complement approach. Just as we use 2's complement representation to deal with negative binary numbers, we can use 10's complement representation to deal with decimal numbers. We leave the development of such a scheme as an exercise for the reader (see Problem 3.19).

3.8 EXAMPLES OF SOLVED PROBLEMS

This section presents some typical problems that the reader may encounter and shows how such problems can be solved.

Example 3.6 **Problem:** Convert the decimal number 14959 into a hexadecimal number.

Solution: An integer is converted into the hexadecimal representation by successive divisions by 16, such that in each step the remainder is a hex digit. To see why this is true, consider a four-digit number $H = h_3h_2h_1h_0$. Its value is

$$V = h_3 \times 16^3 + h_2 \times 16^2 + h_1 \times 16 + h_0$$

If we divide this by 16, we obtain

$$\frac{V}{16} = h_3 \times 16^2 + h_2 \times 16 + h_1 + \frac{h_0}{16}$$

Thus, the remainder gives h_0. Figure 3.42 shows the steps needed to perform the conversion $(14959)_{10} = (3A6F)_{16}$.

Convert $(14959)_{10}$

			Remainder	Hex digit	
14959 ÷ 16	=	934	15	F	LSB
934 ÷ 16	=	58	6	6	
58 ÷ 16	=	3	10	A	
3 ÷ 16	=	0	3	3	MSB

Result is $(3A6F)_{16}$

Figure 3.42 Conversion from decimal to hexadecimal.

Example 3.7

Problem: Convert the decimal fraction 0.8254 into binary representation.

Solution: As indicated in Section 3.7.1, a binary fraction is represented as the bit pattern $B = 0.b_{-1}b_{-2}\cdots b_{-m}$ and its value is

$$V = b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-m} \times 2^{-m}$$

Multiplying this expression by 2 gives

$$b_{-1} + b_{-2} \times 2^{-1} + \cdots + b_{-m} \times 2^{-(m-1)}$$

Here, the left-most term is the first bit to the right of the radix point. The remaining terms constitute another binary fraction which can be manipulated in the same way. Therefore, to convert a decimal fraction into a binary fraction, we multiply the decimal number by 2 and set the computed bit to 0 if the product is less than 1 and set it to 1 if the product is greater than or equal to 1. We repeat this calculation until a sufficient number of bits are obtained to meet the desired accuracy. Note that it may not be possible to represent a decimal fraction with a binary fraction that has exactly the same value. Figure 3.43 shows the required computation that yields $(0.8254)_{10} = (0.11010011\ldots)_2$.

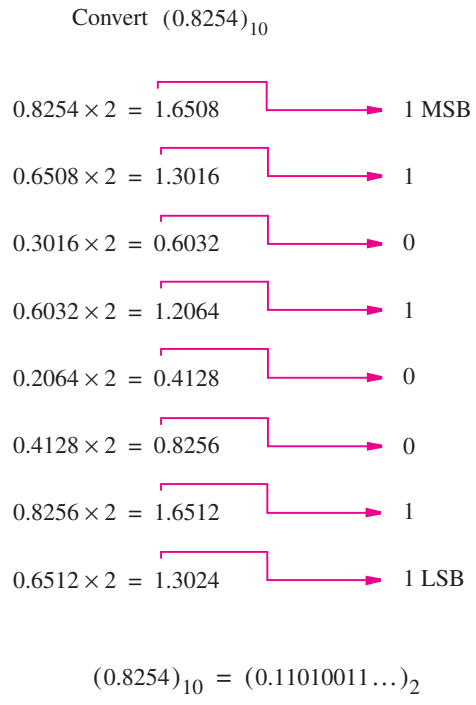


Figure 3.43 Conversion of fractions from decimal to binary.

Example 3.8 **Problem:** Convert the decimal fixed point number 214.45 into a binary fixed point number.

Solution: For the integer part perform successive division by 2 as illustrated in Figure 1.6. For the fractional part perform successive multiplication by 2 as described in Example 3.7. The complete computation is presented in Figure 3.44, producing $(214.45)_{10} = (11010110.0111001\ldots)_2$.

Example 3.9 **Problem:** In computer computations it is often necessary to compare numbers. Two four-bit signed numbers, $X = x_3x_2x_1x_0$ and $Y = y_3y_2y_1y_0$, can be compared by using the subtractor circuit in Figure 3.45, which performs the operation $X - Y$. The three outputs denote the following:

- $Z = 1$ if the result is 0; otherwise $Z = 0$
- $N = 1$ if the result is negative; otherwise $N = 0$
- $V = 1$ if arithmetic overflow occurs; otherwise $V = 0$

Show how Z, N, and V can be used to determine the cases $X = Y, X < Y, X \leq Y, X > Y$, and $X \geq Y$.

Solution: Consider first the case $X < Y$, where the following possibilities may arise:

- If X and Y have the same sign there will be no overflow, hence $V = 0$. Then for both positive and negative X and Y the difference will be negative ($N = 1$).
- If X is negative and Y is positive, the difference will be negative ($N = 1$) if there is no overflow ($V = 0$); but the result will be positive ($N = 0$) if there is overflow ($V = 1$).

Therefore, if $X < Y$ then $N \oplus V = 1$.

The case $X = Y$ is detected by $Z = 1$. Then, $X \leq Y$ is detected by $Z + (N \oplus V) = 1$. The last two cases are just simple inverses: $X > Y$ if $\overline{Z + (N \oplus V)} = 1$ and $X \geq Y$ if $\overline{N \oplus V} = 1$.

Example 3.10 **Problem:** Write Verilog code to specify the circuit in Figure 3.45.

Solution: We can specify the circuit using the approach given in Figure 3.23, as indicated in Figure 3.46. Note that the statement

$$\textbf{assign}\ Z = !S;$$

will produce 1 only if $S = s_3s_2s_1s_0 = 0000$. Thus, it represents the four-bit NOR function

$$Z = \overline{s_3 + s_2 + s_1 + s_0}$$

Instantiating each full adder separately becomes awkward when large circuits are involved, as would be the case if the comparator had 32-bit operands. An alternative is to use the generic specification presented in Figure 3.24, as shown in Figure 3.47.

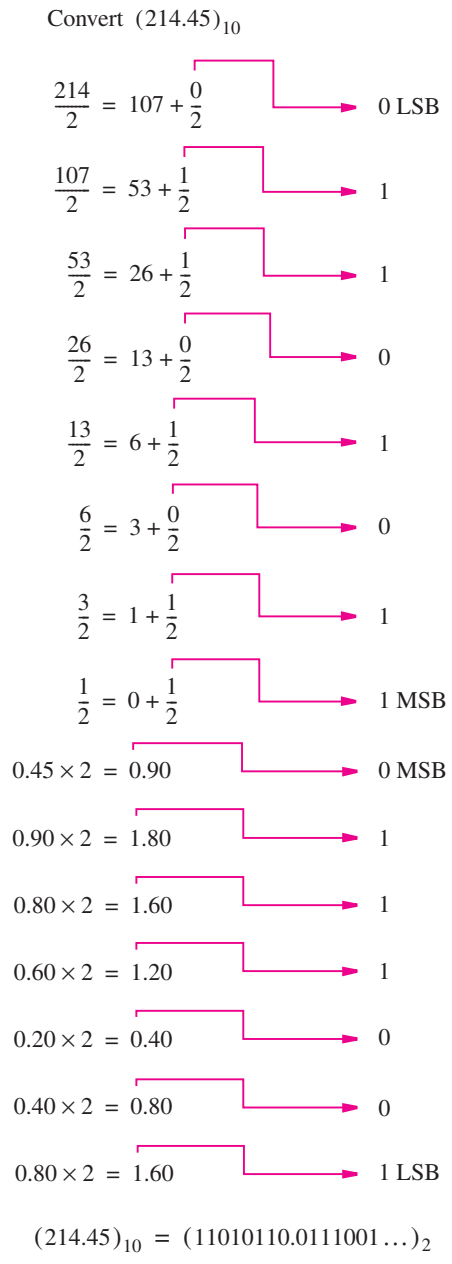


Figure 3.44 Conversion of fixed point numbers from decimal to binary.

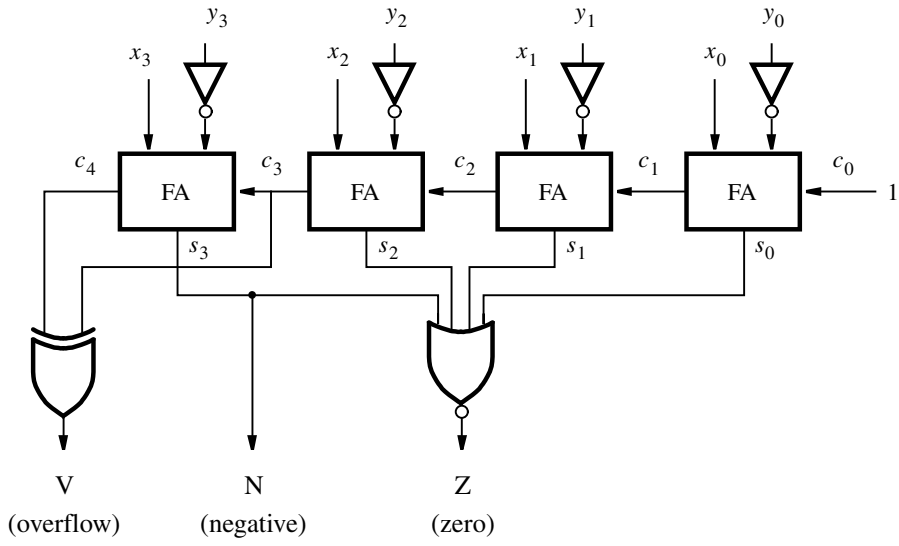


Figure 3.45 A comparator circuit.

```
module comparator (X, Y, V, N, Z);
   input [3:0] X, Y;
   output V, N, Z;
   wire [3:0] S;
   wire [4:1] C;

   fulladd stage0 (1'b1, X[0], ~Y[0], S[0], C[1]);
   fulladd stage1 (C[1], X[1], ~Y[1], S[1], C[2]);
   fulladd stage2 (C[2], X[2], ~Y[2], S[2], C[3]);
   fulladd stage3 (C[3], X[3], ~Y[3], S[3], C[4]);
   assign V = C[4] ^ C[3];
   assign N = S[3];
   assign Z = !S;

endmodule

module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign s = x ^ y ^ Cin;
   assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

Figure 3.46 Structural Verilog code for the comparator circuit.

```
module comparator (X, Y, V, N, Z);
   parameter n = 32;
   input [n –1:0] X, Y;
   output reg V, N, Z;
   reg [n –1:0] S;
   reg [n:0] C;
   integer k;

   always @(X, Y)
   begin
      C[0] = 1'b1;
      for (k = 0; k < n; k = k+1)
      begin
         S[k] = X[k] ^ ~Y[k] ^ C[k];
         C[k+1] = (X[k] & ~Y[k]) | (X[k] & C[k]) | (~Y[k] & C[k]);
      end
      V = C[n] ^ C[n–1];
      N = S[n –1];
      Z = !S;
   end

endmodule
```

Figure 3.47 Generic Verilog code for the comparator circuit.

Example 3.11

Problem: Figure 3.35 depicts a four-bit multiplier circuit. Each row in this circuit consists of four full-adder (FA) blocks connected in a ripple-carry configuration. The delay caused by the carry signals rippling through the rows has a significant impact on the time needed to generate the output product. In an attempt to speed up the circuit, we may use the arrangement in Figure 3.48. Here, the carries in a given row are "saved" and included in the next row at the correct bit position. Then, in the first row the full-adders can be used to add three properly shifted bits of the multiplicand as selected by the multiplier bits. For example, in bit position 2 the three inputs are m_2q_0, m_1q_1, and m_0q_2. In the last row it is still necessary to use the ripple-carry adder. A circuit that consists of an array of full-adders connected in this manner is called a *carry-save* adder array.

What is the total delay of the circuit in Figure 3.48 compared to that of the circuit in Figure 3.35?

Solution: In the circuit in Figure 3.35 the longest path is through the right-most two full-adders in the top row, followed by the two right-most FAs in the second row, and then through all four FAs in the bottom row. Hence this delay is eight times the delay through a full-adder block. In addition, there is the AND-gate delay needed to form the inputs to the first FA in the top row. These combined delays are the critical delay, which determines the speed of the multiplier circuit.

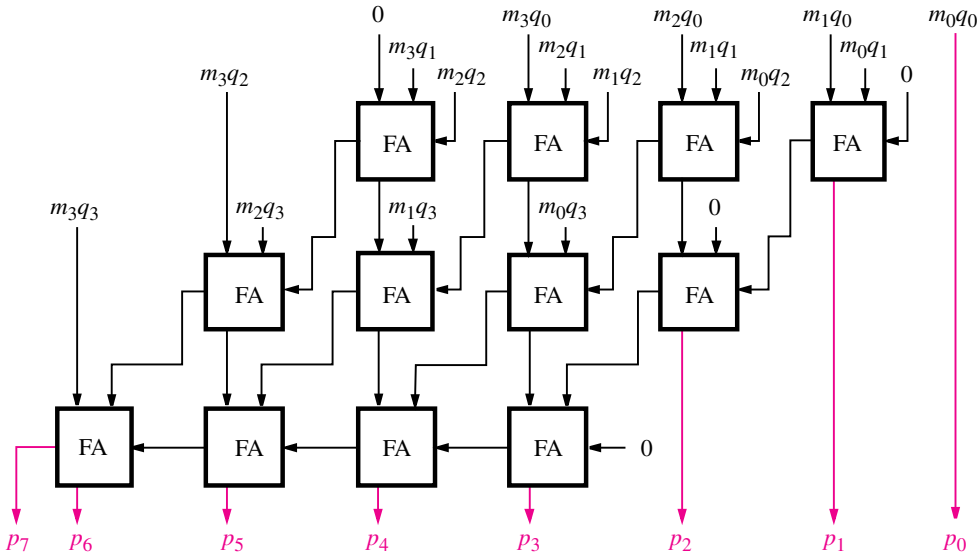


Figure 3.48 Multiplier carry-save array.

In the circuit in Figure 3.48, the longest path is through the right-most FAs in the first and second rows, followed by all four FAs in the bottom row. Therefore, the critical delay is six times the delay through a full-adder block plus the AND-gate delay needed to form the inputs to the first FA in the top row.

PROBLEMS

Answers to problems marked by an asterisk are given at the back of the book.

***3.1** Determine the decimal values of the following unsigned numbers:
(a) $(0111011110)_2$
(b) $(1011100111)_2$
(c) $(3751)_8$
(d) $(A25F)_{16}$
(e) $(F0F0)_{16}$

***3.2** Determine the decimal values of the following 1's complement numbers:
(a) 0111011110
(b) 1011100111
(c) 1111111110

***3.3** Determine the decimal values of the following 2's complement numbers:
(a) 0111011110
(b) 1011100111
(c) 1111111110

***3.4** Convert the decimal numbers 73, 1906, −95, and −1630 into signed 12-bit numbers in the following representations:
(a) Sign and magnitude
(b) 1's complement
(c) 2's complement

3.5 Perform the following operations involving eight-bit 2's complement numbers and indicate whether arithmetic overflow occurs. Check your answers by converting to decimal sign-and-magnitude representation.

00110110	01110101	11011111
+ 01000101	+ 11011110	+ 10111000
00110110	01110101	11010011
− 00101011	− 11010110	− 11101100

3.6 Prove that the XOR operation is associative, which means that $x_i \oplus (y_i \oplus z_i) = (x_i \oplus y_i) \oplus z_i$.

3.7 Show that the circuit in Figure 3.4 implements the full-adder specified in Figure 3.3*a*.

3.8 Prove the validity of the simple rule for finding the 2's complement of a number, which was presented in Section 3.3. Recall that the rule states that scanning a number from right to left, all 0s and the first 1 are copied; then all remaining bits are complemented.

3.9 Prove the validity of the expression Overflow $= c_n \oplus c_{n-1}$ for addition of n-bit signed numbers.

3.10 Verify that a carry-out signal, c_k, from bit position $k-1$ of an adder circuit can be generated as $c_k = x_k \oplus y_k \oplus s_k$, where x_k and y_k are inputs and s_k is the sum bit.

***3.11** Consider the circuit in Figure P3.1. Can this circuit be used as one stage in a ripple-carry adder? Discuss the pros and cons. The operation of transistors shown in the figure is described in Appendix B.

***3.12** Determine the number of gates needed to implement an n-bit carry-lookahead adder, assuming no fan-in constraints. Use AND, OR, and XOR gates with any number of inputs.

***3.13** Determine the number of gates needed to implement an eight-bit carry-lookahead adder assuming that the maximum fan-in for the gates is four.

3.14 In Figure 3.17 we presented the structure of a hierarchical carry-lookahead adder. Show the complete circuit for a four-bit version of this adder, built using 2 two-bit blocks.

3.15 What is the critical delay path in the multiplier in Figure 3.35? What is the delay along this path in terms of the number of gates?

3.16 Write a Verilog module to describe the 4×4 multiplier shown in Figure 3.35. Synthesize a circuit from the code and verify its functional correctness.

***3.17** Consider the Verilog code in Figure P3.2. Given the relationship between the signals IN and OUT, what is the functionality of the circuit described by the code? Comment on whether or not this code represents a good style to use for the functionality that it represents.

3.18 Design a circuit that generates the 9's complement of a BCD digit. Note that the 9's complement of d is $9 - d$.

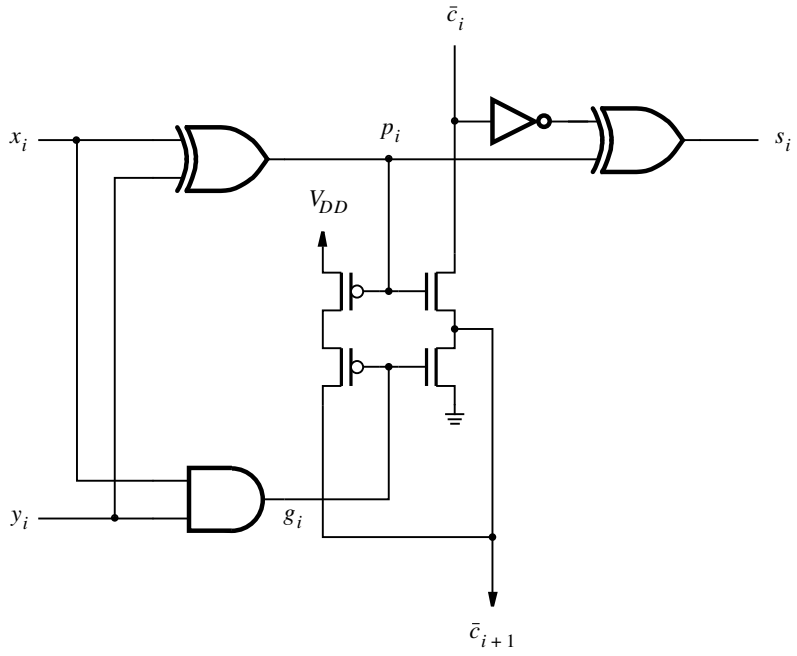


Figure P3.1 Circuit for Problem 3.11.

```
module problem3_17 (IN, OUT);
   input [3:0] IN;
   output reg [3:0] OUT;

   always @(IN)
      if (IN == 4'b0101)   OUT = 4'b0001;
      else if (IN == 4'b0110)   OUT = 4'b0010;
      else if (IN == 4'b0111)   OUT = 4'b0011;
      else if (IN == 4'b1001)   OUT = 4'b0010;
      else if (IN == 4'b1010)   OUT = 4'b0100;
      else if (IN == 4'b1011)   OUT = 4'b0110;
      else if (IN == 4'b1101)   OUT = 4'b0011;
      else if (IN == 4'b1110)   OUT = 4'b0110;
      else if (IN == 4'b1111)   OUT = 4'b1001;
      else   OUT = 4'b0000;

endmodule
```

Figure P3.2 The code for Problem 3.17.

3.19 Derive a scheme for performing subtraction using BCD operands. Show a block diagram for the subtractor circuit. *Hint:* Subtraction can be performed easily if the operands are in the 10's complement (radix complement) representation. In this representation the sign digit is 0 for a positive number and 9 for a negative number.

3.20 Write complete Verilog code for the circuit that you derived in Problem 3.19.

***3.21** Suppose that we want to determine how many of the bits in a three-bit unsigned number are equal to 1. Design the simplest circuit that can accomplish this task.

3.22 Repeat Problem 3.21 for a six-bit unsigned number.

3.23 Repeat Problem 3.21 for an eight-bit unsigned number.

3.24 Show a graphical interpretation of three-digit decimal numbers, similar to Figure 3.11. The left-most digit is 0 for positive numbers and 9 for negative numbers. Verify the validity of your answer by trying a few examples of addition and subtraction.

3.25 In a ternary number system there are three digits: 0, 1, and 2. Figure P3.3 defines a ternary half-adder. Design a circuit that implements this half-adder using binary-encoded signals, such that two bits are used for each ternary digit. Let $A = a_1a_0$, $B = b_1b_0$, and $Sum = s_1s_0$; note that *Carry* is just a binary signal. Use the following encoding: $00 = (0)_3$, $01 = (1)_3$, and $10 = (2)_3$. Minimize the cost of the circuit.

A B	Carry	Sum
0 0	0	0
0 1	0	1
0 2	0	2
1 0	0	1
1 1	0	2
1 2	1	0
2 0	0	2
2 1	1	0
2 2	1	1

Figure P3.3 Ternary half-adder.

3.26 Design a ternary full-adder circuit, using the approach described in Problem 3.25.

3.27 Consider the subtractions $26 - 27 = 99$ and $18 - 34 = 84$. Using the concepts presented in Section 3.3.4, explain how these answers (99 and 84) can be interpreted as the correct signed results of these subtractions.

REFERENCES

1. C. Hamacher, Z. Vranesic, S. Zaky and N. Manjikian, *Computer Organization and Embedded Systems*, 6th ed. (McGraw-Hill: New York, 2011).
2. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design—The Hardware/Software Interface*, 3rd ed. (Morgan Kaufmann: San Francisco, CA, 2004).
3. Institute of Electrical and Electronic Engineers (IEEE), "A Proposed Standard for Floating-Point Arithmetic," *Computer* 14, no. 3 (March 1981), pp. 51–62.