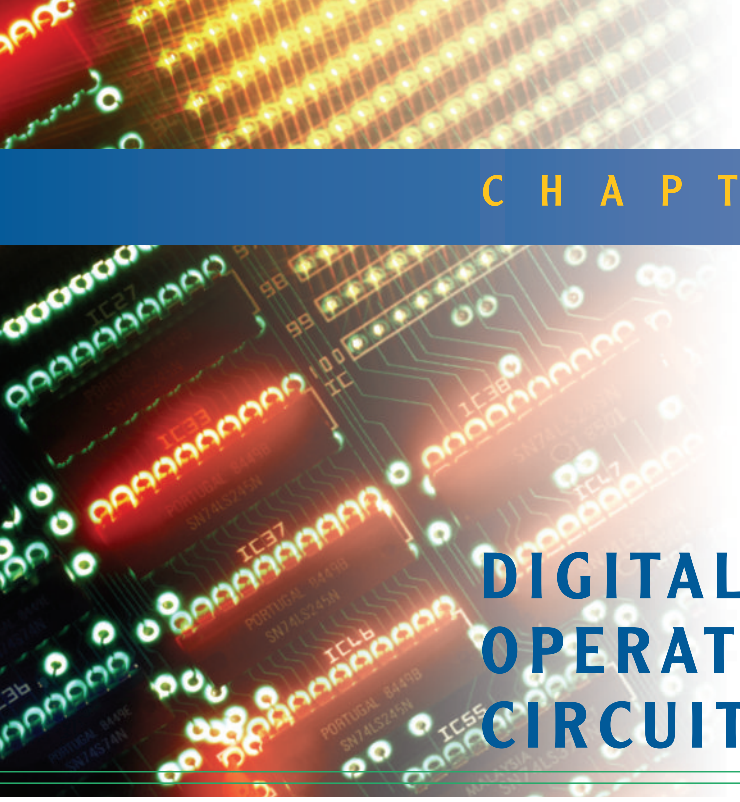


CHAPTER 6

DIGITAL ARITHMETIC: OPERATIONS AND CIRCUITS

OUTLINE

OBJECTIVES

Upon completion of this chapter, you will be able to:

- Perform binary addition, subtraction, multiplication, and division on two binary numbers.
- Add and subtract hexadecimal numbers.
- Know the difference between binary addition and OR addition.
- Compare the advantages and disadvantages among three different systems of representing signed binary numbers.
- Manipulate signed binary numbers using the 2's-complement system.
- Understand the BCD addition process.
- Describe the basic operation of an arithmetic/logic unit.
- Employ full adders in the design of parallel binary adders.
- Cite the advantages of parallel adders with the look-ahead carry feature.
- Explain the operation of a parallel adder/subtractor circuit.
- Use an ALU integrated circuit to perform various logic and arithmetic operations on input data.
- Analyze troubleshooting case studies of adder/subtractor circuits.
- Use HDL forms of standard TTL parts from libraries to implement more complicated circuits.
- Use the Boolean equation form of description to perform operations on entire sets of bits.
- Apply software engineering techniques to expand the capacity of a hardware description.

INTRODUCTION

Digital computers and calculators perform the various arithmetic operations on numbers that are represented in binary form. The subject of digital arithmetic can be a very complex one if we want to understand all the various methods of computation and the theory behind them. Fortunately, this level of knowledge is not required by most technicians, at least not until they become experienced computer programmers. Our approach in this chapter will be to concentrate on those basic principles that are necessary for understanding how digital machines (i.e., computers) perform the basic arithmetic operations.

First, we will see how the various arithmetic operations are performed on binary numbers using "pencil and paper," and then we will study the

actual logic circuits that perform these operations in a digital system. Finally, we will learn how to describe these simple circuits using HDL techniques. Several methods of expanding the capacity of these circuits will also be covered. The focus will be on the fundamentals of HDL, using arithmetic circuits as an example. The powerful capability of HDL combined with PLD hardware will provide the basis for further study, design, and experimentation with much more sophisticated arithmetic circuits in more advanced courses.

6-1 BINARY ADDITION

The addition of two binary numbers is performed in exactly the same manner as the addition of decimal numbers. In fact, binary addition is simpler because there are fewer cases to learn. Let us first review decimal addition:

$$
\begin{array}{rrrl}
 & & \downarrow & \quad\downarrow \\
3 & 7 & 6 & \quad \text{LSD} \\
+4 & 6 & 1 & \\
\hline
8 & 3 & 7 &
\end{array}
$$

The least-significant-digit (LSD) position is operated on first, producing a sum of 7. The digits in the second position are then added to produce a sum of 13, which produces a **carry** of 1 into the third position. This produces a sum of 8 in the third position.

The same general steps are followed in binary addition. However, only four cases can occur in adding the two binary digits (bits) in any position. They are:

$$
\begin{array}{r}
0 + 0 = 0 \phantom{= 0 + \text{carry of 1 into next position}} \\
1 + 0 = 1 \phantom{= 0 + \text{carry of 1 into next position}} \\
1 + 1 = 10 = 0 + \text{carry of 1 into next position} \\
1 + 1 + 1 = 11 = 1 + \text{carry of 1 into next position}
\end{array}
$$

The last case occurs when the two bits in a certain position are 1 and there is a carry from the previous position. Here are several examples of the addition of two binary numbers (decimal equivalents are in parentheses):

$$
\begin{array}{rr}
 & 011\ (3) \\
+ & 110\ (6) \\
\hline
 & 1001\ (9)
\end{array}
\qquad\qquad
\begin{array}{rr}
 & 1001\ (9) \\
+ & 1111\ (15) \\
\hline
 & 11000\ (24)
\end{array}
\qquad\qquad
\begin{array}{rr}
 & 11.011\ (3.375) \\
+ & 10.110\ (2.750) \\
\hline
 & 110.001\ (6.125)
\end{array}
$$

It is not necessary to consider the addition of more than two binary numbers at a time because in all digital systems the circuitry that actually performs the addition can handle only two numbers at a time. When more than two numbers are to be added, the first two are added together and then their sum is added to the third number, and so on. This is not a serious drawback because modern digital computers can typically perform an addition operation in several nanoseconds.

Addition is the most important arithmetic operation in digital systems. As we shall see, the operations of subtraction, multiplication, and division as they are performed in most modern digital computers and calculators actually use only addition as their basic operation.

REVIEW QUESTION

1. Add the following pairs of binary numbers.
   (a) 10110 + 00111
   (b) 011.101 + 010.010
   (c) 10001111 + 00000001

6-2 REPRESENTING SIGNED NUMBERS

In digital computers, the binary numbers are represented by a set of binary storage devices (e.g., flip-flops). Each device represents one bit. For example, a six-bit FF register can store binary numbers ranging from 000000 to 111111 (0 to 63 in decimal). This represents the *magnitude* of the number. Because most digital computers and calculators handle negative as well as positive numbers, some means is required for representing the *sign* of the number (+ or −). This is usually done by adding to the number another bit called the **sign bit**. In general, the common convention is that a 0 in the sign bit represents a positive number and a 1 in the sign bit represents a negative number. This is illustrated in Figure 6-1. Register *A* contains the bits 0110100. The 0 in the leftmost bit (A_6) is the sign bit that represents +. The other six bits are the magnitude of the number 110100_2, which is equal to 52 in decimal. Thus, the number stored in the *A* register is +52. Similarly, the number stored in the *B* register is −52 because the sign bit is 1, representing −.

The sign bit is used to indicate the positive or negative nature of the stored binary number. The numbers in Figure 6-1 consist of a sign bit and six magnitude bits. The magnitude bits are the true binary equivalent of the decimal value being represented. This is called the **sign-magnitude system** for representing signed binary numbers.

Although the sign-magnitude system is straightforward, calculators and computers do not normally use it because the circuit implementation is more complex than in other systems. The most commonly used system for representing signed binary numbers is the **2's-complement system**. Before we see how this is done, we must first see how to form the 1's complement and 2's complement of a binary number.

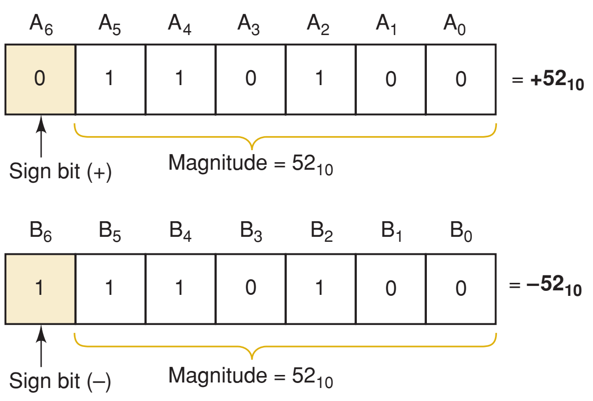


FIGURE 6-1 Representation of signed numbers in sign-magnitude form.

1's-Complement Form

The 1's complement of a binary number is obtained by changing each 0 to a 1 and each 1 to a 0. In other words, change each bit in the number to its complement. The process is shown below.

```
1 0 1 1 0 1  original binary number
↓↓↓↓↓↓
0 1 0 0 1 0  complement each bit to form 1's complement
```

Thus, we say that the 1's complement of 101101 is 010010.

2's Complement Form

The 2's complement of a binary number is formed by taking the 1's complement of the number and adding 1 to the least-significant-bit position. The process is illustrated below for $101101_2 = 45_{10}$.

```
   1 0 1 1 0 1     binary equivalent of 45
   0 1 0 0 1 0     complement each bit to form 1's complement
+            1     add 1 to form 2's complement
  ------------
   0 1 0 0 1 1     2's complement of original binary number
```

Thus, we say that 010011 is the 2's complement representation of 101101.

Here's another example of converting a binary number to its 2's-complement representation:

```
   1 0 1 1 0 0     original binary number
   0 1 0 0 1 1     1's complement
+            1     add 1
  ------------
   0 1 0 1 0 0     2's complement of original number
```

Representing Signed Numbers Using 2's Complement

The 2's-complement system for representing signed numbers works like this:

- If the number is positive, the magnitude is represented in its true binary form, and a sign bit of 0 is placed in front of the MSB. This is shown in Figure 6-2 for $+45_{10}$.
- If the number is negative, the magnitude is represented in its 2's-complement form, and a sign bit of 1 is placed in front of the MSB. This is shown in Figure 6-2 for -45_{10}.

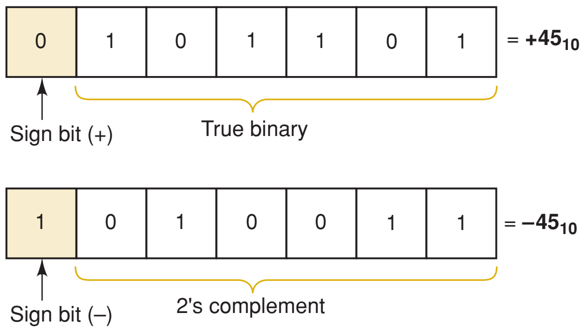


FIGURE 6-2 Representation of signed numbers in the 2's-complement system.

The 2's-complement system is used to represent signed numbers because, as we shall see, it allows us to perform the operation of subtraction by actually performing addition. This is significant because it means that a digital computer can use the same circuitry both to add and to subtract, thereby realizing a saving in hardware.

EXAMPLE 6-1

Represent each of the following signed decimal numbers as a signed binary number in the 2's-complement system. Use a total of five bits, including the sign bit.

(a) $+13$ (b) -9 (c) $+3$ (d) -2 (e) -8

Solution

(a) The number is positive, so the magnitude (13) will be represented in its true-magnitude form, that is, $13 = 1101_2$. Attaching the sign bit of 0, we have

$$\mathbf{+13 = \underset{\uparrow\ \text{sign bit}}{0}1101}$$

(b) The number is negative, so the magnitude (9) must be represented in 2's-complement form:

$$\begin{array}{rrl} 9_{10} = & 1001_2 & \\ & 0110 & \text{(1's complement)} \\ + & \underline{\quad 1} & \text{(add 1 to LSB)} \\ & 0111 & \text{(2's complement)} \end{array}$$

When we attach the sign bit of 1, the complete signed number becomes

$$\mathbf{-9 = \underset{\uparrow\ \text{sign bit}}{1}0111}$$

The procedure we have just followed required two steps. First, we determined the 2's complement of the magnitude, and then we attached the sign bit. This can be accomplished in one step if we include the sign bit in the 2's-complement process. For example, to find the representation for -9, we start with the representation for $+9$, *including the sign bit,* and we take the 2's complement of it in order to obtain the representation for -9.

$$\begin{array}{rrl} +9 = & 01001 & \\ & 10110 & \text{(1's complement of each bit including sign bit)} \\ + & \underline{\quad 1} & \text{(add 1 to LSB)} \\ -9 = & 10111 & \text{(2's-complement representation of } -9) \end{array}$$

The result is, of course, the same as before.

(c) The decimal value 3 can be represented in binary using only two bits. However, the problem statement requires a four-bit magnitude preceded by a sign bit. Thus, we have

$$+3_{10} = 00011$$

In many situations the number of bits is fixed by the size of the registers that will be holding the binary numbers, so that 0s may have to be added in order to fill the required number of bit positions.

(d) Start by writing +2 using five bits:

$$\begin{array}{rrl} +2 = & 00010 & \\ & 11101 & \text{(1's complement)} \\ + & \underline{1} & \text{(add 1)} \\ -2 = & 11110 & \text{(2's-complement representation of } -2\text{)} \end{array}$$

(e) Start with +8:

$$\begin{array}{rrl} +8 = & 01000 & \\ & 10111 & \text{(complement each bit)} \\ + & \underline{1} & \text{(add 1)} \\ -8 = & 11000 & \text{(2's-complement representation of } -8\text{)} \end{array}$$

Sign Extension

Example 6-1 required that we use a total of five bits to represent the signed numbers. The size of a register (*number of flip-flops*) determines the number of binary digits that are stored for each number. Most digital systems today store numbers in registers sized in even multiples of four bits. In other words, the storage registers will be made up of 4, 8, 12, 16, 32, or 64 bits. In a system that stores eight-bit numbers, seven bits represent the magnitude and the MSB represents the sign. If we need to store a positive five-bit number in an eight-bit register, it makes sense to simply add leading zeros. The MSB (sign bit) is still 0, indicating a positive value.

$$\underbrace{000}_{\text{appended leading 0s}}\underbrace{0\ 1001}_{\text{binary value for 9}}$$

What happens when we try to store five-bit negative numbers in an eight-bit register? In the previous section we found that the five-bit, 2's-complement binary representation for −9 is 10111.

1 0111

If we appended leading 0s, this would no longer be a negative number in eight-bit format. The proper way to extend a negative number is to append leading 1's. Thus, the value stored for negative 9 is

111 1 0111

2's complement magnitude

sign in five-bit format

sign extension to eight-bit format

Negation

Negation is the operation of converting a positive number to its negative equivalent or a negative number to its positive equivalent. When signed binary numbers are represented in the 2's-complement system, negation is performed simply by performing the 2's-complement operation. To illustrate, let's start with +9 in eight-bit binary form. Its signed representation is 00001001. If we take its 2's complement we get 11110111, which represents the signed value −9. Likewise, we can start with the representation of −9, which is 11110111, and take its 2's complement to get 00001001, which represents +9. These steps are diagrammed below.

Start with	00001001	+9
2's complement (negate)	11110111	−9
negate again	00001001	+9

Thus, we negate a signed binary number by 2's-complementing it.

This negation changes the number to its equivalent of opposite sign. We used negation in steps (d) and (e) of Example 6-1 to convert positive numbers to their negative equivalents.

EXAMPLE 6-2

Each of the following numbers is a five-bit signed binary number in the 2's-complement system. Determine the decimal value in each case:

(a) 01100 (b) 11010 (c) 10001

Solution

(a) The sign bit is 0, so the number is *positive* and the other four bits represent the true magnitude of the number. That is, $1100_2 = 12_{10}$. Thus, the decimal number is +12.

(b) The sign bit of 11010 is a 1, so we know that the number is negative, but we can't tell what the magnitude is. We can find the magnitude by negating (2's-complementing) the number to convert it to its positive equivalent.

$$\begin{array}{rl} 11010 & \text{(original negative number)} \\ 00101 & \text{(1's complement)} \\ +\underline{1} & \text{(add 1)} \\ 00110 & (+6) \end{array}$$

Because the result of the negation is 00110 = +6, the original number 11010 must be equivalent to −6.

(c) Follow the same procedure as in (b):

$$\begin{array}{rl} 10001 & \text{(original negative number)} \\ 01110 & \text{(1's complement)} \\ +\underline{1} & \text{(add 1)} \\ 01111 & (+15) \end{array}$$

Thus, 10001 = −15.

Special Case in 2's-Complement Representation

Whenever a signed number has a 1 in the sign bit and all 0s for the magnitude bits, its decimal equivalent is -2^N, where N is the number of bits in the *magnitude.* For example,

$$\begin{aligned} 1000 &= -2^3 = -8 \\ 10000 &= -2^4 = -16 \\ 100000 &= -2^5 = -32 \end{aligned}$$

and so on. Notice that in this special case, taking the 2's complement of these numbers produces the value we started with because we are at the negative limit of the range of numbers that can be represented by this many bits. If we extend the sign of these special numbers, the normal negation procedure works fine. For example, extending the number 1000 (−8) to 11000 (five-bit negative 8) and taking its 2's complement we get 01000 (8), which is the magnitude of the negative number.

Thus, we can state that the complete range of values that can be represented in the 2's-complement system having N magnitude bits is

$$-2^N \text{ to } +(2^N - 1)$$

There are a total of 2^{N+1} different values, *including* zero.

For example, Table 6-1 lists all signed numbers that can be represented in four bits using the 2's-complement system (note there are three magnitude bits, so $N = 3$). Note that the sequence starts at $-2^N = -2^3 = -8_{10} = 1000_2$ and proceeds upward to $+(2^N - 1) = +2^3 - 1 = +7_{10} = 0111_2$ by adding 0001 at each step as in an up counter.

TABLE 6-1

Decimal Value	Signed Binary Using 2's Complement
$+7 = 2^3 - 1$	0111
+6	0110
+5	0101
+4	0100
+3	0011
+2	0010
+1	0001
0	0000
−1	1111
−2	1110
−3	1101
−4	1100
−5	1011
−6	1010
−7	1001
$-8 = -2^3$	1000

EXAMPLE 6-3

What is the range of *unsigned* decimal values that can be represented in a byte?

Solution

Recall that a byte is eight bits. We are interested in unsigned numbers here, so there is no sign bit, and all of the eight bits are used for the magnitude. Therefore, the values will range from

$$00000000_2 = 0_{10}$$

to

$$11111111_2 = 255_{10}$$

This is a total of 256 different values, which we could have predicted because $2^8 = 256$.

EXAMPLE 6-4

What is the range of *signed* decimal values that can be represented in a byte?

Solution

Because the MSB is to be used as the sign bit, there are seven bits for the magnitude. The largest negative value is

$$10000000_2 = -2^7 = -128_{10}$$

The largest positive value is

$$01111111_2 = +2^7 - 1 = +127_{10}$$

Thus, the range is −128 to +127; this is a total of 256 different values, including zero. Alternatively, because there are seven magnitude bits ($N = 7$), then there are $2^{N+1} = 2^8 = 256$ different values.

EXAMPLE 6-5

A certain computer is storing the following two signed numbers in its memory using the 2's-complement system:

$$00011111_2 = +31_{10}$$
$$11110100_2 = -12_{10}$$

While executing a program, the computer is instructed to convert each number to its opposite sign; that is, change the +31 to −31 and change the −12 to +12. How will it do this?

Solution

This is the negation operation whereby a signed number can have its polarity changed simply by performing the 2's-complement operation on the *complete* number, including the sign bit. The computer circuitry will take the signed number from memory, find its 2's complement, and put the result back in memory.

REVIEW QUESTIONS

1. Represent each of the following values as an eight-bit signed number in the 2's-complement system.
   (a) +13 (b) −7 (c) −128
2. Each of the following is a signed binary number in the 2's-complement system. Determine the decimal equivalent for each.
   (a) 100011 (b) 1000000 (c) 01111110
3. What range of signed decimal values can be represented in 12 bits (including the sign bit)?
4. How many bits are required to represent decimal values ranging from −50 to +50?
5. What is the largest negative decimal value that can be represented by a two-byte number?
6. Perform the 2's-complement operation on each of the following.
   (a) 10000 (b) 10000000 (c) 1000
7. Define the negation operation.

6-3 ADDITION IN THE 2's-COMPLEMENT SYSTEM

We will now investigate how the operations of addition and subtraction are performed in digital machines that use the 2's-complement representation for negative numbers. In the various cases to be considered, it is important to note that the sign bit of each number is operated on in the same manner as the magnitude bits.

Case I: Two Positive Numbers. The addition of two positive numbers is straightforward. Consider the addition of +9 and +4:

$$\begin{array}{rcll}
+9 \rightarrow & 0 & 1001 & \quad\text{(augend)}\\
+4 \rightarrow & 0 & 0100 & \quad\text{(addend)}\\ \hline
 & 0 & 1101 & \quad(\text{sum} = +13)\\
 & \uparrow & \text{sign bits} &
\end{array}$$

Note that the sign bits of the **augend** and the **addend** are both 0 and the sign bit of the sum is 0, indicating that the sum is positive. Also note that the augend and the addend are made to have the same number of bits. This must *always* be done in the 2's-complement system.

Case II: Positive Number and Smaller Negative Number. Consider the addition of +9 and −4. Remember that the −4 will be in its 2's-complement form. Thus, +4 (00100) must be converted to −4 (11100).

$$\begin{array}{rcll}
 & \downarrow & \text{sign bits} & \\
+9 \rightarrow & 0 & 1001 & \quad\text{(augend)}\\
-4 \rightarrow & 1 & 1100 & \quad\text{(addend)}\\ \hline
\cancel{1} & 0 & 0101 & \\
\uparrow & \multicolumn{3}{l}{\text{This carry is disregarded; the result is 00101 (sum = +5).}}
\end{array}$$

In this case, the sign bit of the addend is 1. Note that the sign bits also participate in the addition process. In fact, a carry is generated in the last position of addition. *This carry is always disregarded,* so that the final sum is 00101, which is equivalent to +5.

Case III: Positive Number and Larger Negative Number. Consider the addition of −9 and +4:

$$\begin{array}{rll} -9 \rightarrow & 10111 & \\ +4 \rightarrow & 00100 & \\ \hline & 11011 & (\text{sum} = -5) \\ & \uparrow\!\text{—— negative sign bit} & \end{array}$$

The sum here has a sign bit of 1, indicating a negative number. Because the sum is negative, it is in 2's-complement form, so that the last four bits, 1011, actually represent the 2's complement of the sum. To find the true magnitude of the sum, we must negate (2's-complement) 11011; the result is 00101 = +5. Thus, 11011 represents −5.

Case IV: Two Negative Numbers

$$\begin{array}{rl} -9 \rightarrow & 10111 \\ -4 \rightarrow & 11100 \\ \hline \cancel{1} & 10011 \\ \uparrow & \uparrow\!\text{— sign bit} \\ & \text{This carry is disregarded; the result is 10011 (sum} = -13). \end{array}$$

This final result is again negative and in 2's-complement form with a sign bit of 1. Negating (2's-complementing) this result produces 01101 = +13.

Case V: Equal and Opposite Numbers

$$\begin{array}{rl} -9 \rightarrow & 10111 \\ +9 \rightarrow & 01001 \\ \hline 0 \quad \cancel{1} & 00000 \\ \uparrow & \text{—— Disregard; the result is 00000 (sum} = +0). \end{array}$$

The result is obviously +0, as expected.

REVIEW QUESTIONS

Assume the 2's-complement system for both questions.

1. *True or false:* Whenever the sum of two signed binary numbers has a sign bit of 1, the magnitude of the sum is in 2's-complement form.
2. Add the following pairs of signed numbers. Express the sum as a signed binary number and as a decimal number.
   (a) 100111 + 111011 (b) 100111 + 011001

6-4 SUBTRACTION IN THE 2's-COMPLEMENT SYSTEM

The subtraction operation using the 2's-complement system actually involves the operation of addition and is really no different from the various cases for addition considered in Section 6-3. When subtracting one binary number (the **subtrahend**) from another binary number (the **minuend**), use the following procedure:

1. *Negate the subtrahend.* This will change the subtrahend to its equivalent value of opposite sign.
2. *Add this to the minuend.* The result of this addition will represent the *difference* between the subtrahend and the minuend.

Once again, as in all 2's-complement arithmetic operations, it is necessary that both numbers have the same number of bits in their representations.

Let us consider the case where +4 is to be subtracted from +9.

```
   minuend (+9) →   01001
subtrahend (+4) →   00100
```

Negate the subtrahend to produce 11100, which represents −4. Now add this to the minuend.

```
   01001   (+9)
 + 11100   (−4)
 -------
 1 00101   (+5)
 ↑
 └ Disregard, so the result is 00101 = +5.
```

When the subtrahend is changed to its 2's complement, it actually becomes −4, so that we are *adding* −4 and +9, which is the same as subtracting +4 from +9. This is the same as case II of Section 6-3. Any subtraction operation, then, actually becomes one of addition when the 2's-complement system is used. This feature of the 2's-complement system has made it the most widely used of the methods available because it allows addition and subtraction to be performed by the same circuitry.

Here's another example showing +9 subtracted from −4:

```
   11100   (−4)
 − 01001   (+9)
```

Negate the subtrahend (+9) to produce 10111 (−9) and add this to the minuend (−4).

```
   11100   (−4)
 + 10111   (−9)
 -------
 1 10011   (−13)
 ↑
 └ Disregard
```

The reader should verify the results of using the above procedure for the following subtractions: (a) +9 − (−4); (b) −9 − (+4); (c) −9 − (−4); (d) +4 − (−4). Remember that when the result has a sign bit of 1, it is negative and in 2's-complement form.

Arithmetic Overflow

In each of the previous addition and subtraction examples, the numbers that were added consisted of a sign bit and four magnitude bits. The answers also consisted of a sign bit and four magnitude bits. Any carry into the sixth bit position was disregarded. In all of the cases considered, the magnitude of the answer was small enough to fit into four bits. Let's look at the addition of +9 and +8.

$$\begin{array}{r|c|l} +9 \rightarrow & 0 & 1001 \\ +8 \rightarrow & 0 & 1000 \\ \hline & \boxed{1} & \underbrace{0001} \\ \text{incorrect sign} \nearrow & & \nwarrow \text{incorrect magnitude} \end{array}$$

The answer has a negative sign bit, which is obviously incorrect because we are adding two positive numbers. The answer should be +17, but the magnitude 17 requires more than four bits and therefore *overflows* into the sign-bit position. This **overflow** condition can occur only when two positive or two negative numbers are being added, and it always produces an incorrect result. Overflow can be detected by checking to see that the sign bit of the result is the same as the sign bits of the numbers being added.

Subtraction in the 2's-complement system is performed by negating the minuend and *adding* it to the subtrahend, so overflow can occur only when the minuend and subtrahend have different signs. For example, if we are subtracting −8 from +9, the −8 is negated to become +8 and is added to +9, just as shown above, and overflow produces an erroneous negative result because the magnitude is too large.

A computer will have a special circuit to detect any overflow condition when two numbers are added or subtracted. This detection circuit will signal the computer's control unit that overflow has occurred and the result is incorrect. We will examine such a circuit in an end-of-chapter problem.

Number Circles and Binary Arithmetic

The concept of signed arithmetic and overflow can be illustrated by taking the numbers from Table 6-1 and "bending" them into a number circle as shown in Figure 6-3. Notice that there are two ways to look at this circle. It can be thought of as a circle of unsigned numbers (as shown in the outer ring) with minimum value 0 and maximum 15, or as signed 2's-complement numbers (as shown in the inner ring) with maximum value 7 and minimum −8. To add using a number circle, simply start at the value of the augend and

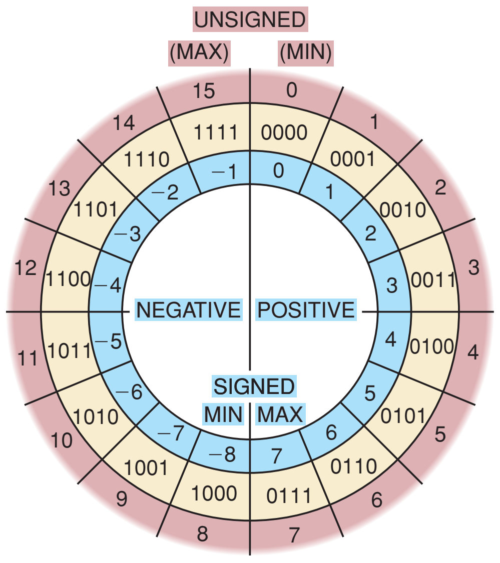


FIGURE 6-3 A four-bit number circle.

advance around the number circle clockwise by the number of spaces in the addend. For example, to add 2 + 3, start at 2 (0010) and then advance clockwise three more spaces to arrive at 5 (0101). Overflow occurs when the sum is too big to fit into four-bit signed format, meaning we have exceeded the maximum value of 7. On the number circle this is indicated when adding two positive values causes us to cross the line between 0111 (max positive) and 1000 (max negative).

The number circle can also illustrate how 2's-complement subtraction really works. For example, let's perform the subtraction of 5 from 3. Of course, we know the answer is −2, but let's run the problem through the number circle. First we start at the number 3 (0011) on the number circle. The most apparent way to subtract is to move *counterclockwise* around the circle five spaces, which lands us on the number 1110 (−2). The less obvious operation that illustrates 2's-complement arithmetic is to add −5 to the number 3. Negative five (the 2's complement of 0101) is 1011 which, interpreted as an unsigned binary number, represents the value 11 (eleven) in decimal. Start at the number 3 (0011) and move clockwise around the circle 11 spaces and you will once again find yourself arriving at the number 1110 (−2), which is the correct result.

Any subtraction operation between four-bit numbers of opposite sign that produces a result greater than 7 or less than −8 is an overflow of the four-bit format and results in an incorrect answer. For example, 3 minus −6 should produce the answer 9, but moving clockwise six spaces from 3 lands us on the signed number −7: an overflow condition has occurred, giving us an incorrect answer.

REVIEW QUESTIONS

1. Perform the subtraction on the following pairs of signed numbers using the 2's-complement system. Express the results as signed binary numbers and as decimal values.

   (a) 01001 − 11010 (b) 10010 − 10011

2. How can arithmetic overflow be detected when signed numbers are being added? Subtracted?

6-5 MULTIPLICATION OF BINARY NUMBERS

The multiplication of binary numbers is done in the same manner as the multiplication of decimal numbers. The process is actually simpler because the multiplier digits are either 0 or 1 and so we are always multiplying by 0 or 1 and no other digits. The following example illustrates for unsigned binary numbers:

$$
\begin{array}{r l}
1001 & \leftarrow \text{multiplicand} = 9_{10} \\
\underline{1011} & \leftarrow \text{multiplier} = 11_{10} \\
\left.\begin{array}{r} 1001 \\ 1001 \\ 0000 \\ \underline{1001} \end{array}\right\} & \text{partial products} \\
1100011 \;\} & \text{final product} = 99_{10}
\end{array}
$$

In this example the multiplicand and the multiplier are in true binary form and no sign bits are used. The steps followed in the process are exactly the same as in decimal multiplication. First, the LSB of the multiplier is examined; in our example, it is a 1. This 1 multiplies the multiplicand to produce 1001, which is written down as the first partial product. Next, the second bit of the multiplier is examined. It is a 1, and so 1001 is written for the second partial product. Note that this second partial product is *shifted* one place to the left relative to the first one. The third bit of the multiplier is 0, and 0000 is written as the third partial product; again, it is shifted one place to the left relative to the previous partial product. The fourth multiplier bit is 1, and so the last partial product is 1001 shifted again one position to the left. The four partial products are then summed to produce the final product.

Most digital machines can add only two binary numbers at a time. For this reason, the partial products formed during multiplication cannot all be added together at the same time. Instead, they are added together two at a time; that is, the first is added to the second, their sum is added to the third, and so on. This process is now illustrated for the example above:

```
      {    1001   ← first partial product
  Add {   1001    ← second partial product shifted left
          -------
      {   11011   ← sum of first two partial products
  Add {  0000     ← third partial product shifted left
         --------
      {  011011   ← sum of first three partial products
  Add { 1001      ← fourth partial product shifted left
        ---------
        1100011   ← sum of four partial products, which
                    equals final total product
```

Multiplication in the 2's-Complement System

In computers that use the 2's-complement representation, multiplication is carried on in the manner described above, provided that both the multiplicand and the multiplier are put in true binary form. If the two numbers to be multiplied are positive, they are already in true binary form and are multiplied as they are. The resulting product is, of course, positive and is given a sign bit of 0. When the two numbers are negative, they will be in 2's-complement form. The 2's complement of each is taken to convert it to a positive number, and then the two numbers are multiplied. The product is kept as a positive number and is given a sign bit of 0.

When one of the numbers is positive and the other is negative, the negative number is first converted to a positive magnitude by taking its 2's complement. The product will be in true-magnitude form. However, the product must be negative because the original numbers are of opposite sign. Thus, the product is then changed to 2's-complement form and is given a sign bit of 1.

REVIEW QUESTION

1. Multiply the unsigned numbers 0111 and 1110.

6-6 BINARY DIVISION

The process for dividing one binary number (the *dividend*) by another (the *divisor*) is the same as that followed for decimal numbers, that which we usually refer to as "long division." The actual process is simpler in binary because when we are checking to see how many times the divisor "goes into" the dividend, there are only two possibilities, 0 or 1. To illustrate, consider the following simple division examples:

$$\begin{array}{rr} & 0011 \\ 11 & \overline{)1001} \\ & \underline{011} \\ & 0011 \\ & \underline{11} \\ & 0 \end{array} \qquad\qquad \begin{array}{rr} & 0010.1 \\ 100 & \overline{)1010.0} \\ & \underline{100} \\ & 100 \\ & \underline{100} \\ & 0 \end{array}$$

In the first example, we have 1001_2 divided by 11_2, which is equivalent to $9 \div 3$ in decimal. The resulting quotient is $0011_2 = 3_{10}$. In the second example, 1010_2 is divided by 100_2, or $10 \div 4$ in decimal. The result is $0010.1_2 = 2.5_{10}$.

In most modern digital machines, the subtractions that are part of the division operation are usually carried out using 2's-complement subtraction, that is, taking the 2's complement of the subtrahend and then adding.

The division of signed numbers is handled in the same way as multiplication. Negative numbers are made positive by complementing, and the division is then carried out. If the dividend and the divisor are of opposite sign, the resulting quotient is changed to a negative number by taking its 2's-complement and is given a sign bit of 1. If the dividend and the divisor are of the same sign, the quotient is left as a positive number and is given a sign bit of 0.

6-7 BCD ADDITION

In Chapter 2, we stated that many computers and calculators use the BCD code to represent decimal numbers. Recall that this code takes *each* decimal digit and represents it by a four-bit code ranging from 0000 to 1001. The addition of decimal numbers that are in BCD form can be best understood by considering the two cases that can occur when two decimal digits are added.

Sum Equals 9 or Less

Consider adding 5 and 4 using BCD to represent each digit:

$$\begin{array}{rrl} 5 & 0101 & \leftarrow \text{BCD for 5} \\ +\underline{4} & +\ \underline{0100} & \leftarrow \text{BCD for 4} \\ 9 & 1001 & \leftarrow \text{BCD for 9} \end{array}$$

The addition is carried out as in normal binary addition, and the sum is 1001, which is the BCD code for 9. As another example, take 45 added to 33:

$$\begin{array}{rrl} 45 & 0100\ \ 0101 & \leftarrow \text{BCD for 45} \\ +\underline{33} & +\ \underline{0011\ \ 0011} & \leftarrow \text{BCD for 33} \\ 78 & 0111\ \ 1000 & \leftarrow \text{BCD for 78} \end{array}$$

In this example, the four-bit codes for 5 and 3 are added in binary to produce 1000, which is BCD for 8. Similarly, adding the second-decimal-digit positions produces 0111, which is BCD for 7. The total is 01111000, which is the BCD code for 78.

In the examples above, none of the sums of the pairs of decimal digits exceeded 9; therefore, *no decimal carries were produced.* For these cases, the BCD addition process is straightforward and is actually the same as binary addition.

Sum Greater than 9

Consider the addition of 6 and 7 in BCD:

$$\begin{array}{rrl}
6 & 0110 & \leftarrow \text{BCD for 6} \\
\underline{+7} & +\ \underline{0111} & \leftarrow \text{BCD for 7} \\
+13 & 1101 & \leftarrow \text{invalid code group for BCD}
\end{array}$$

The sum 1101 does not exist in the BCD code; it is one of the six forbidden or invalid four-bit code groups. This has occurred because the sum of the two digits exceeds 9. Whenever this occurs, the sum must be corrected by the addition of six (0110) to take into account the skipping of the six invalid code groups:

$$\begin{array}{rrl}
 & 0110 & \leftarrow \text{BCD for 6} \\
 & +\ \underline{0111} & \leftarrow \text{BCD for 7} \\
 & 1101 & \leftarrow \text{invalid sum} \\
 & \underline{0110} & \leftarrow \text{add 6 for correction} \\
\underbrace{0001}_{1} & \underbrace{0011}_{3} & \leftarrow \text{BCD for 13}
\end{array}$$

As shown above, 0110 is added to the invalid sum and produces the correct BCD result. Note that with the addition of 0110, a carry is produced in the second decimal position. This addition must be performed whenever the sum of the two decimal digits is greater than 9.

As another example, take 47 plus 35 in BCD:

$$\begin{array}{rrrl}
47 & 0100 & 0111 & \leftarrow \text{BCD for 47} \\
+\underline{35} & +\ \underline{0011} & \underline{0101} & \leftarrow \text{BCD for 35} \\
82 & 0111 & 1100 & \leftarrow \text{invalid sum in first digit} \\
 & 1 \leftarrow & 0110 & \leftarrow \text{add 6 to correct} \\
 & \underbrace{1000}_{8} & \underbrace{0010}_{2} & \leftarrow \text{correct BCD sum}
\end{array}$$

The addition of the four-bit codes for the 7 and 5 digits results in an invalid sum and is corrected by adding 0110. Note that this generates a carry of 1, which is carried over to be added to the BCD sum of the second-position digits.

Consider the addition of 59 and 38 in BCD:

$$\begin{array}{rrrl}
 & 1 & & \\
59 & 0101 & 1001 & \leftarrow \text{BCD for 59} \\
+\underline{38} & +\ \underline{0011} & \underline{1000} & \leftarrow \text{BCD for 38} \\
97 & 1001 & 0001 & \leftarrow \text{perform addition} \\
 & & \underline{0110} & \leftarrow \text{add 6 to correct} \\
 & \underbrace{1001}_{9} & \underbrace{0111}_{7} & \quad \text{BCD for 97}
\end{array}$$

Here, the addition of the least significant digits (LSDs) produces a sum of 17 = 10001. This generates a carry into the next digit position to be added to the codes for 5 and 3. Since 17 > 9, a correction factor of 6 must be added to the LSD sum. Addition of this correction does not generate a carry; the carry was already generated in the original addition.

To summarize the BCD addition procedure:

1. Using ordinary binary addition, add the BCD code groups for each digit position.
2. For those positions where the sum is 9 or less, no correction is needed. The sum is in proper BCD form.
3. When the sum of two digits is greater than 9, a correction of 0110 should be added to that sum to get the proper BCD result. This case always produces a carry into the next digit position, either from the original addition (step 1) or from the correction addition.

The procedure for BCD addition is clearly more complicated than straight binary addition. This is also true of the other BCD arithmetic operations. Readers should perform the addition of 275 + 641. Then check the correct procedure below.

$$\begin{array}{rrrrrl}
275 & & 0010 & 0111 & 0101 & \leftarrow \text{BCD for 275} \\
+\underline{641} & + & \underline{0110} & \underline{0100} & \underline{0001} & \leftarrow \text{BCD for 641} \\
916 & & 1000 & 1011 & 0110 & \leftarrow \text{perform addition} \\
 & + & \underline{} & \underline{0110} & \underline{} & \leftarrow \text{add 6 to correct second digit} \\
 & & 1001 & 0001 & 0110 & \leftarrow \text{BCD for 916}
\end{array}$$

BCD Subtraction

The process of subtracting BCD numbers is more difficult than addition. It involves a complement-then-add procedure similar to the 2's-complement method. We do not cover it in this book.

REVIEW QUESTIONS

1. How can you tell when a correction is needed in BCD addition?
2. Represent 135_{10} and 265_{10} in BCD and then perform BCD addition. Check your work by converting the result back to decimal.

6-8 HEXADECIMAL ARITHMETIC

Hex numbers are used extensively in machine-language computer programming and in conjunction with computer memories (i.e., addresses). When working in these areas, you will encounter situations where hex numbers must be added or subtracted.

Hex Addition

Addition of hexadecimal numbers is done in much the same way as decimal addition, as long as you remember that the largest hex digit is F instead of 9. The following procedure is suggested:

1. Add the two hex digits in decimal, mentally inserting the decimal equivalent for those digits larger than 9.

2. If the sum is 15 or less, it can be directly expressed as a hex digit.
3. If the sum is greater than or equal to 16, subtract 16 and carry a 1 to the next digit position.

The following examples will illustrate the procedure.

EXAMPLE 6-6

Add the hex numbers 58 and 24.

Solution

$$\begin{array}{r} 58 \\ +\underline{24} \\ 7\text{C} \end{array}$$

Adding the LSDs (8 and 4) produces 12, which is C in hex. There is no carry into the next digit position. Adding 5 and 2 produces 7.

EXAMPLE 6-7

Add the hex numbers 58 and 4B.

Solution

$$\begin{array}{r} 58 \\ +\underline{4\text{B}} \\ \text{A}3 \end{array}$$

Start by adding 8 and B, mentally substituting decimal 11 for B. This produces a sum of 19. Because 19 is greater than 16, subtract 16 to get 3; write down the 3 and carry a 1 into the next position. This carry is added to the 5 and 4 to produce a sum of 10_{10}, which is then converted to hexadecimal A.

EXAMPLE 6-8

Add 3AF to 23C.

Solution

$$\begin{array}{r} 3\text{AF} \\ +\underline{23\text{C}} \\ 5\text{EB} \end{array}$$

The sum of F and C is considered as $15 + 12 = 27_{10}$. Because this is greater than 16, subtract 16 to get 11_{10}, which is hexadecimal B, and carry a 1 into the second position. Add this carry to A and 3 to obtain E. There is no carry into the MSD position.

Hex Subtraction

Remember that hex numbers are just an efficient way to represent binary numbers. Thus, we can subtract hex numbers using the same method we used for binary numbers. The 2's complement of the hex subtrahend will be taken and then *added* to the minuend, and any carry out of the MSD position will be disregarded.

How do we find the 2's complement of a hex number? One way is to convert it to binary, take the 2's complement of the binary equivalent, and then convert it back to hex. This process is illustrated below.

$$\begin{array}{ccccl} & & 73A & & \leftarrow \text{hex number}\\ 0111 & & 0011 & & 1010 \quad \leftarrow \text{convert to binary}\\ 1000 & & 1100 & & 0110 \quad \leftarrow \text{take 2's complement}\\ & & 8C6 & & \leftarrow \text{convert back to hex} \end{array}$$

There is a quicker procedure: subtract *each* hex digit from F; then add 1. Let's try this for the same hex number from the example above.

$$\begin{array}{ccc l} F & F & F & \\ -\underline{7} & -\underline{3} & -\underline{A} & \left.\right\} \leftarrow \text{subtract each digit from F}\\ 8 & C & 5 & \\ & & +1 & \leftarrow \text{add 1}\\ \hline 8 & C & 6 & \leftarrow \text{hex equivalent of 2's complement} \end{array}$$

Try either of the procedures above on the hex number E63. The correct result for the 2's complement is 19D.

CALCULATOR HINT

On a hex calculator, you can subtract the hex digits from a string of F's and then add one as we just demonstrated, or you can add one to the string of all F's and then subtract. For example, adding 1 to FFF_{16} yields 1000_{16}. On the hex calculator enter:

1000 − 73A =   The answer is 8C6

EXAMPLE 6-9

Subtract $3A5_{16}$ from 592_{16}.

Solution

First, convert the subtrahend (3A5) to its 2's-complement form by using either method presented above. The result is C5B. Then add this to the minuend (592):

$$\begin{array}{r} 592\\ +\ \underline{C5B}\\ \cancel{1}1ED \end{array}$$

↑ Disregard carry.

Ignore the carry out of the MSD addition; the result is 1ED. We can prove that this is correct by adding 1ED to 3A5 and checking to see that it equals 592_{16}.

Hex Representation of Signed Numbers

The data stored in a microcomputer's internal working memory or on a hard disk or CD ROM are typically stored in bytes (groups of eight bits). The data

TABLE 6-2

Hex Address	Stored Binary Data	Hex Value	Decimal Value
4000	00111010	3A	+58
4001	11100101	E5	−29
4002	01010111	57	+87
4003	10000000	80	−128

byte stored in a particular memory location is often expressed in hexadecimal because it is more efficient and less error-prone than expressing it in binary. When the data consist of *signed* numbers, it is helpful to be able to recognize whether a hex value represents a positive or a negative number. For example, Table 6-2 lists the data stored in a small segment of memory starting at address 4000.

Each memory location stores a single byte (eight bits), which is the binary equivalent of a signed decimal number. The table also shows the hex equivalent of each byte. For a negative data value, the sign bit (MSB) of the binary number will be a 1; this will always make the MSD of the hex number 8 or greater. When the data value is positive, the sign bit will be a 0, and the MSD of the hex number will be 7 or less. The same holds true no matter how many digits are in the hex number. *When the MSD is 8 or greater, the number being represented is negative; when the MSD is 7 or less, the number is positive.*

REVIEW QUESTIONS

1. Add 67F + 2A4.
2. Subtract 67F − 2A4.
3. Which of the following hex numbers represent positive values: 2F, 77EC, C000, 6D, FFFF?

6-9 ARITHMETIC CIRCUITS

One essential function of most computers and calculators is the performance of arithmetic operations. These operations are all performed in the arithmetic/logic unit of a computer, where logic gates and flip-flops are combined so that they can add, subtract, multiply, and divide binary numbers. These circuits perform arithmetic operations at speeds that are not humanly possible. Typically, an addition operation will take less than 100 ns.

We will now study some of the basic arithmetic circuits that are used to perform the arithmetic operations discussed earlier. In some cases, we will go through the actual design process, even though the circuits may be commercially available in integrated-circuit form, to provide more practice in the use of the techniques learned in Chapter 4.

Arithmetic/Logic Unit

All arithmetic operations take place in the **arithmetic/logic unit (ALU)** of a computer. Figure 6-4 is a block diagram showing the major elements included in a typical ALU. The main purpose of the ALU is to accept binary

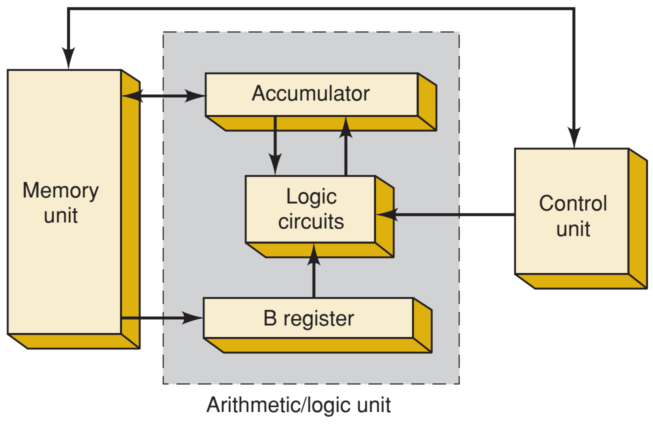


FIGURE 6-4 Functional parts of an ALU.

data that are stored in the memory and to execute arithmetic and logic operations on these data according to instructions from the control unit.

The arithmetic/logic unit contains at least two flip-flop registers: the *B register* and the **accumulator register**. It also contains combinational logic, which performs the arithmetic and logic operations on the binary numbers that are stored in the *B* register and the accumulator. A typical sequence of operations may occur as follows:

1. The control unit receives an instruction (from the memory unit) specifying that a number stored in a particular memory location (address) is to be added to the number presently stored in the accumulator register.
2. The number to be added is transferred from memory to the *B* register.
3. The number in the *B* register and the number in the accumulator register are added together in the logic circuits (upon command from the control unit). The resulting sum is then sent to the accumulator to be stored.
4. The new number in the accumulator can remain there so that another number can be added to it or, if the particular arithmetic process is finished, it can be transferred to memory for storage.

These steps should make it apparent how the accumulator register derives its name. This register "accumulates" the sums that occur when performing successive additions between new numbers acquired from memory and the previously accumulated sum. In fact, for any arithmetic problem containing several steps, the accumulator usually contains the results of the intermediate steps as they are completed as well as the final result when the problem is finished.

6-10 PARALLEL BINARY ADDER

Computers and calculators perform the addition operation on two binary numbers at a time, where each binary number can have several binary digits. Figure 6-5 illustrates the addition of two five-bit numbers. The **augend** is stored in the accumulator register; that is, the accumulator contains five FFs, storing the values 10101 in successive FFs. Similarly, the **addend**, the number that is to be added to the augend, is stored in the *B* register (in this case, 00111).

The addition process starts by adding the least significant bits (LSBs) of the augend and addend. Thus, 1 + 1 = 10, which means that the *sum* for that position is 0, with a *carry* of 1.

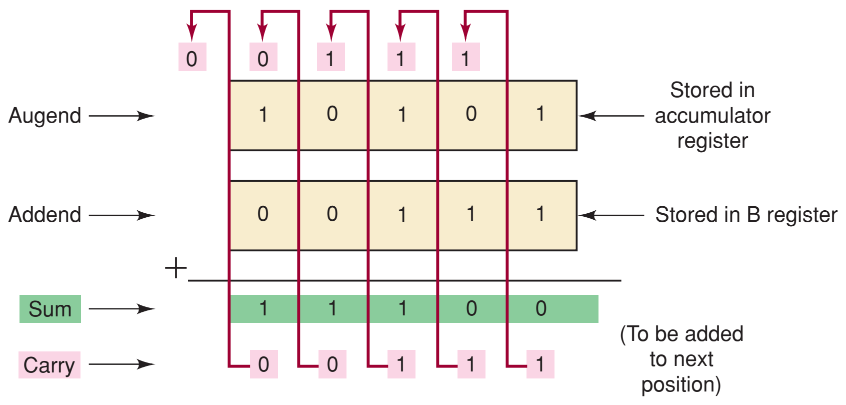


FIGURE 6-5 Typical binary addition process.

This carry must be added to the next position along with the augend and addend bits in that position. Thus, in the second position, 1 + 0 + 1 = 10, which is again a sum of 0 and a carry of 1. This carry is added to the next position together with the augend and addend bits in that position, and so on, for the remaining positions, as shown in Figure 6-5.

At each step in this addition process, we are performing the addition of three bits: the augend bit, the addend bit, and a carry bit from the previous position. The result of the addition of these three bits produces two bits: a *sum* bit, and a *carry* bit that is to be added to the next position. It should be clear that the same process is followed for each bit position. Thus, if we can design a logic circuit that can duplicate this process, then all we have to do is to use the identical circuit for each of the bit positions. This is illustrated in Figure 6-6.

In this diagram, variables A_4, A_3, A_2, A_1, and A_0 represent the bits of the augend that are stored in the accumulator (which is also called the *A* register). Variables B_4, B_3, B_2, B_1, and B_0 represent the bits of the addend stored in the *B* register. Variables C_4, C_3, C_2, C_1, and C_0 represent the carry bits into the corresponding positions. Variables S_4, S_3, S_2, S_1, S_0 are the sum output bits for each position. Corresponding bits of the augend and addend are fed to a logic circuit called a **full adder (FA)**, along with a carry bit from the previous position. For example, bits A_1 and B_1 are fed into full adder 1 along

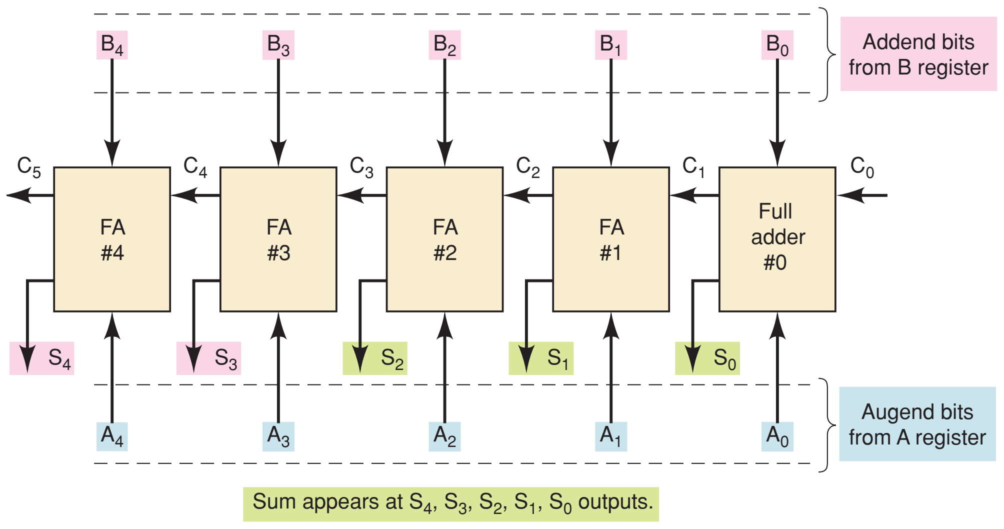


FIGURE 6-6 Block diagram of a five-bit parallel adder circuit using full adders.

with C_1, which is the carry bit produced by the addition of the A_0 and B_0 bits. Bits A_0 and B_0 are fed into full adder 0 along with C_0. A_0 and B_0 are the LSBs of the augend and addend, so it appears that C_0 would always have to be 0 because there can be no carry into that position. We shall see, however, that there will be situations when C_0 can also be 1.

The full-adder circuit used in each position has three inputs: an A bit, a B bit, and a C bit. It also produces two outputs: a sum bit and a carry bit. For example, full adder 0 has inputs A_0, B_0, and C_0, and it produces outputs S_0 and C_1. Full adder 1 had A_1, B_1, and C_1 as inputs and S_1 and C_2 as outputs, and so on. This arrangement is repeated for as many positions as there are in the augend and addend. Although this illustration is for five-bit numbers, in modern computers the numbers usually range from 8 to 64 bits.

The arrangement in Figure 6-6 is called a **parallel adder** because all of the bits of the augend and addend are present and are fed into the adder circuits *simultaneously*. This means that the additions in each position are taking place at the same time. This is different from how we add on paper, taking each position one at a time starting with the LSB. Clearly, parallel addition is extremely fast. More will be said about this later.

REVIEW QUESTIONS

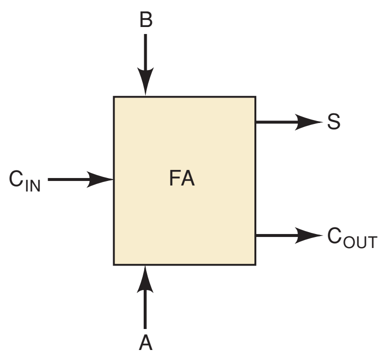
1. How many inputs does a full adder have? How many outputs?
2. Assume the following input levels in Figure 6-6: $A_4A_3A_2A_1A_0 = 01001$; $B_4B_3B_2B_1B_0 = 00111$; $C_0 = 0$.
   (a) What are the logic levels at the outputs of FA #2?
   (b) What is the logic level at the C_5 output?

6-11 DESIGN OF A FULL ADDER

Now that we know the function of the full adder, we can design a logic circuit that will perform this function. First, we must construct a truth table showing the various input and output values for all possible cases. Figure 6-7 shows the truth table having three inputs, A, B, and C_{IN}, and two outputs, S and C_{OUT}. There are eight possible cases for the three inputs, and for each

FIGURE 6-7 Truth table for a full-adder circuit.

Augend bit input	Addend bit input	Carry bit input	Sum bit output	Carry bit output
A	B	C_{IN}	S	C_{OUT}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



case the desired output values are listed. For example, consider the case $A = 1$, $B = 0$, and $C_{IN} = 1$. The full adder (FA) must add these bits to produce a sum (S) of 0 and a carry (C_{OUT}) of 1. The reader should check the other cases to be sure they are understood.

Because there are two outputs, we will design the circuitry for each output individually, starting with the S output. The truth table shows that there are four cases where S is to be a 1. Using the sum-of-products method, we can write the expression for S as

$$S = \overline{A}\,\overline{B}C_{IN} + \overline{A}B\overline{C}_{IN} + A\overline{B}\,\overline{C}_{IN} + ABC_{IN} \tag{6-1}$$

We can now try to simplify this expression by factoring. Unfortunately, none of the terms in the expression has two variables in common with any of the other terms. However, $\overline{A}$ can be factored from the first two terms, and A can be factored from the last two terms:

$$S = \overline{A}(\overline{B}C_{IN} + B\overline{C}_{IN}) + A(\overline{B}\,\overline{C}_{IN} + BC_{IN})$$

The first term in parentheses should be recognized as the exclusive-OR combination of B and C_{IN}, which can be written as $B \oplus C_{IN}$. The second term in parentheses should be recognized as the exclusive-NOR of B and C_{IN}, which can be written as $\overline{B \oplus C_{IN}}$. Thus, the expression for S becomes

$$S = \overline{A}(B \oplus C_{IN}) + A(\overline{B \oplus C_{IN}})$$

If we let $X = B \oplus C_{IN}$, this can be written as

$$S = \overline{A} \cdot X + A \cdot \overline{X} = A \oplus X$$

which is simply the exclusive-OR of A and X. Replacing the expression for X, we have

$$S = A \oplus [B \oplus C_{IN}] \tag{6-2}$$

Consider now the output C_{OUT} in the truth table of Figure 6-7. We can write the sum-of-products expression for C_{OUT} as follows:

$$C_{OUT} = \overline{A}BC_{IN} + A\overline{B}C_{IN} + AB\overline{C}_{IN} + ABC_{IN}$$

This expression can be simplified by factoring. We will employ the trick introduced in Chapter 4, whereby we will use the ABC_{IN} term *three* times because it has common factors with each of the other terms. Hence,

$$\begin{aligned} C_{OUT} &= BC_{IN}(\overline{A} + A) + AC_{IN}(\overline{B} + B) + AB(\overline{C}_{IN} + C_{IN}) \\ &= BC_{IN} + AC_{IN} + AB \end{aligned} \tag{6-3}$$

This expression cannot be simplified further.

Expressions (6-2) and (6-3) can be implemented as shown in Figure 6-8. Several other implementations can be used to produce the same expressions for S and C_{OUT}, none of which has any particular advantage over those shown. The complete circuit with inputs A, B, and C_{IN} and outputs S and C_{OUT} represents the full adder. Each of the FAs in Figure 6-6 contains this same circuitry (or its equivalent).

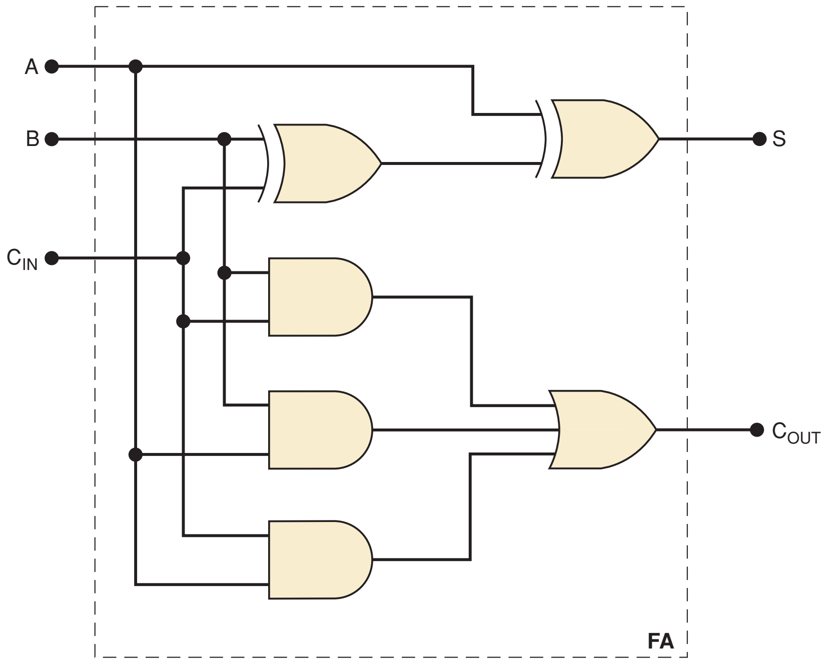


FIGURE 6-8 Complete circuitry for a full adder.

K-Map Simplification

We simplified the expressions for S and C_{OUT} using algebraic methods. The K-map method can also be used. Figure 6-9(a) shows the K map for the S output. This map has no adjacent 1s, and so there are no pairs or quads to loop. Thus, the expression for S cannot be simplified using the K map. This points out a limitation of the K-map method compared with the algebraic method. We were able to simplify the expression for S through factoring and the use of XOR and XNOR operations.

The K map for the C_{OUT} output is shown in Figure 6-9(b). The three pairs that are looped will produce the same expression obtained from the algebraic method.

Half Adder

The FA operates on three inputs to produce a sum and carry output. In some cases, a circuit is needed that will add only two input bits, to produce

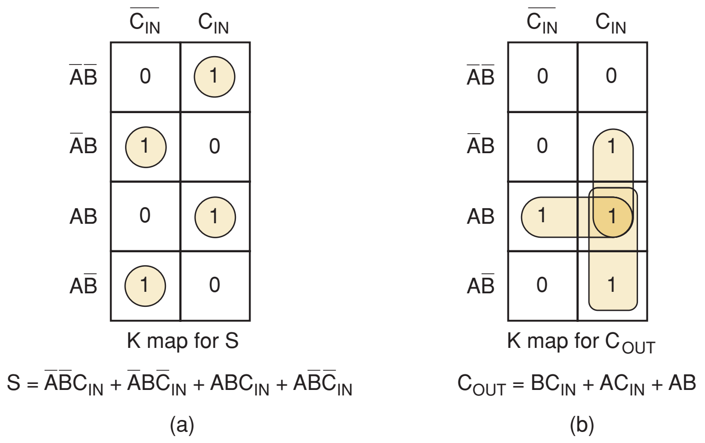

$\overline{C}_{IN}$ C_{IN}
$\overline{A}\overline{B}$ 0 1
$\overline{A}B$ 1 0
AB 0 1
$A\overline{B}$ 1 0
K map for S
$S = \overline{A}\overline{B}C_{IN} + \overline{A}B\overline{C}_{IN} + ABC_{IN} + A\overline{B}\overline{C}_{IN}$
(a)
$\overline{C}_{IN}$ C_{IN}
$\overline{A}\overline{B}$ 0 0
$\overline{A}B$ 0 1
AB 1 1
$A\overline{B}$ 0 1
K map for C_{OUT}
$C_{OUT} = BC_{IN} + AC_{IN} + AB$
(b)

FIGURE 6-9 K mappings for the full-adder outputs.

a sum and carry output. An example would be the addition of the LSB position of two binary numbers where there is no carry input to be added. A special logic circuit can be designed to take *two* input bits, A and B, and to produce sum (S) and carry (C_{OUT}) outputs. This circuit is called a **half adder (HA)**. Its operation is similar to that of an FA except that it operates on only two bits. We shall leave the design of the HA as an exercise at the end of the chapter.

6-12 COMPLETE PARALLEL ADDER WITH REGISTERS

In a computer, the numbers that are to be added are stored in FF registers. Figure 6-10 shows the diagram of a four-bit parallel adder, including the storage registers. The augend bits A_3 through A_0 are stored in the accumulator (A register); the addend bits B_3 through B_0 are stored in the B register. Each of these registers is made up of D flip-flops for easy transfer of data.

The contents of the A register (i.e., the binary number stored in A_3 through A_0) is added to the contents of the B register by the four FAs, and the sum is produced at outputs S_3 through S_0. C_4 is the carry out of the fourth FA, and it can be used as the carry input to a fifth FA, or as an *overflow* bit to indicate that the sum exceeds 1111.

Note that the sum outputs are connected to the D inputs of the A register. This will allow the sum to be parallel-transferred into the A register on the positive-going transition (PGT) of the TRANSFER pulse. In this way, the sum can be stored in the A register.

Also note that the D inputs of the B register are coming from the computer's memory, so that binary numbers from memory will be parallel-transferred into the B register on the PGT of the LOAD pulse. In most computers, there is also provision for parallel-transferring binary numbers from memory into the accumulator (A register). For simplicity, the circuitry necessary for performing this transfer is not shown in this diagram; it will be addressed in an end-of-chapter exercise.

Finally, note that the A register outputs are available for transfer to other locations such as another computer register or the computer's memory. This will make the adder circuit available for a new set of numbers.

Register Notation

Before we go through the complete process of how this circuit adds two binary numbers, it will be helpful to introduce some notation that makes it easy to describe the contents of a register and data transfer operations.

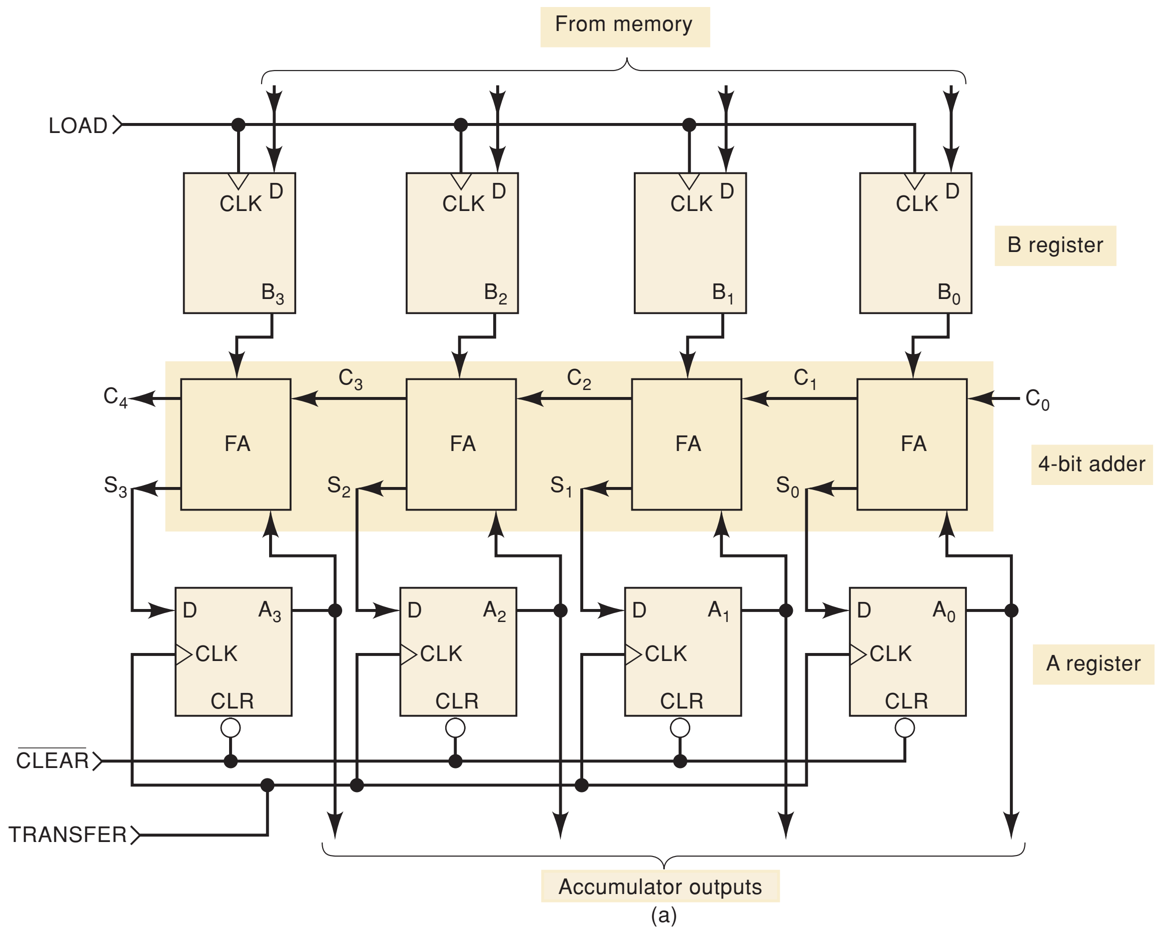
Whenever we want to give the levels that are present at each FF in a register or at each output of a group of outputs, we will use brackets, as illustrated below:

$$[A] = 1011$$

This is the same as saying that $A_3 = 1$, $A_2 = 0$, $A_1 = 1$, $A_0 = 1$. In other words, think of $[A]$ as representing "the contents of register A."

Whenever we want to indicate the transfer of data to or from a register, we will use an arrow, as illustrated below:

$$[B] \rightarrow [A]$$



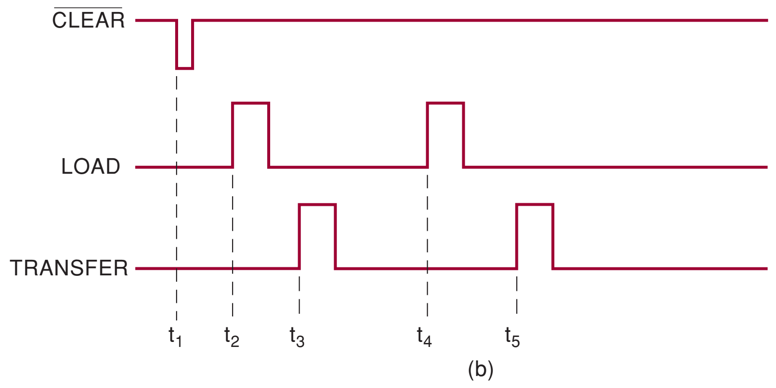


FIGURE 6-10 (a) Complete four-bit parallel adder with registers; (b) signals used to add binary numbers from memory and store their sum in the accumulator.

This means that the contents of the *B* register have been transferred to the *A* register. The old contents of the *A* register will be lost as a result of this operation, and the *B* register will be unchanged. This type of notation is very common, especially in data books describing microprocessor and microcontroller operations. In many ways, it is very similar to the notation used to refer to bit-array data objects using hardware description languages.

Sequence of Operations

We will now describe the process by which the circuit of Figure 6-10 will add the binary numbers 1001 and 0101. Assume that $C_0 = 0$; that is, there is no carry into the LSB position.

1. $[A] = 0000$. A $\overline{\text{CLEAR}}$ pulse is applied to the asynchronous inputs $(\overline{CLR})$ of each FF in register A. This occurs at time t_1.
2. $[M] \rightarrow [B]$. This first binary number is transferred from memory (M) to the B register. In this case, the binary number 1001 is loaded into register B on the PGT of the LOAD pulse at t_2.
3. $[S]^* \rightarrow [A]$. With $[B] = 1001$ and $[A] = 0000$, the full adders produce a sum of 1001; that is, $[S] = 1001$. These sum outputs are transferred into the A register on the PGT of the TRANSFER pulse at t_3. This makes $[A] = 1001$.
4. $[M] \rightarrow [B]$. The second binary number, 0101, is transferred from memory into the B register on the PGT of the second LOAD pulse at t_4. This makes $[B] = 0101$.
5. $[S] \rightarrow [A]$. With $[B] = 0101$ and $[A] = 1001$, the FAs produce $[S] = 1110$. These sum outputs are transferred into the A register when the second TRANSFER pulse occurs at t_5. Thus, $[A] = 1110$.
6. At this point, the sum of the two binary numbers is present in the accumulator. In most computers, the contents of the accumulator, $[A]$, will usually be transferred to the computer's memory so that the adder circuit can be used for a new set of numbers. The circuitry that performs this $[A] \rightarrow [M]$ transfer is not shown in Figure 6-10.

REVIEW QUESTIONS

1. Suppose that four different four-bit numbers are to be taken from memory and added by the circuit of Figure 6-10. How many $\overline{\text{CLEAR}}$ pulses will be needed? How many TRANSFER pulses? How many LOAD pulses?
2. Determine the contents of the A register after the following sequence of operations: $[A] = 0000$, $[0110] \rightarrow [B]$, $[S] \rightarrow [A]$, $[1110] \rightarrow [B]$, $[S] \rightarrow [A]$.

6-13 CARRY PROPAGATION

The parallel adder of Figure 6-10 performs additions at a relatively high speed because it adds the bits from each position simultaneously. However, its speed is limited by an effect called **carry propagation** or **carry ripple**, which can best be explained by considering the following addition:

$$\begin{array}{r} 0111 \\ +\ \underline{0001} \\ 1000 \end{array}$$

Addition of the LSB position produces a carry into the second position. This carry, when added to the bits of the second position, produces a carry into

*Even though S is not a register, we will use $[S]$ to represent the group of S outputs.

the third position. The latter carry, when added to the bits of the third position, produces a carry into the last position. The key point to notice in this example is that the sum bit generated in the *last* position (MSB) depended on the carry that was generated by the addition in the *first* position (LSB).

Looking at this from the viewpoint of the circuit of Figure 6-10, S_3 out of the last full adder depends on C_1 out of the first full adder. But the C_1 signal must pass through three FAs before it produces S_3. What this means is that the S_3 output will not reach its correct value until C_1 has propagated through the intermediate FAs. This represents a time delay that depends on the propagation delay produced in each FA. For example, if each FA has a propagation delay of 40 ns, then S_3 will not reach its correct level until 120 ns after C_1 is generated. This means that the add command pulse cannot be applied until 160 ns after the augend and addend numbers are present in the FF registers (the extra 40 ns is due to the delay of the LSB full adder, which generates C_1).

Obviously, the situation becomes much worse if we extend the adder circuitry to add a greater number of bits. If the adder were handling 32-bit numbers, the carry propagation delay could be 1280 ns $=$ 1.28 μs. The add pulse could not be applied until at least 1.28 μs after the numbers were present in the registers.

This magnitude of delay is prohibitive for high-speed computers. Fortunately, logic designers have come up with several ingenious schemes for reducing this delay. One of the schemes, called **look-ahead carry**, utilizes logic gates to look at the lower-order bits of the augend and addend to see if a higher-order carry is to be generated. For example, it is possible to build a logic circuit with B_2, B_1, B_0, A_2, A_1, and A_0 as inputs and C_3 as an output. This logic circuit would have a shorter delay than is obtained by the carry propagation through the FAs. This scheme requires a large amount of extra circuitry but is necessary to produce high-speed adders. The extra circuitry is not a significant consideration with the present use of integrated circuits. Many high-speed adders available in integrated-circuit form utilize the look-ahead carry or a similar technique for reducing overall propagation delays.

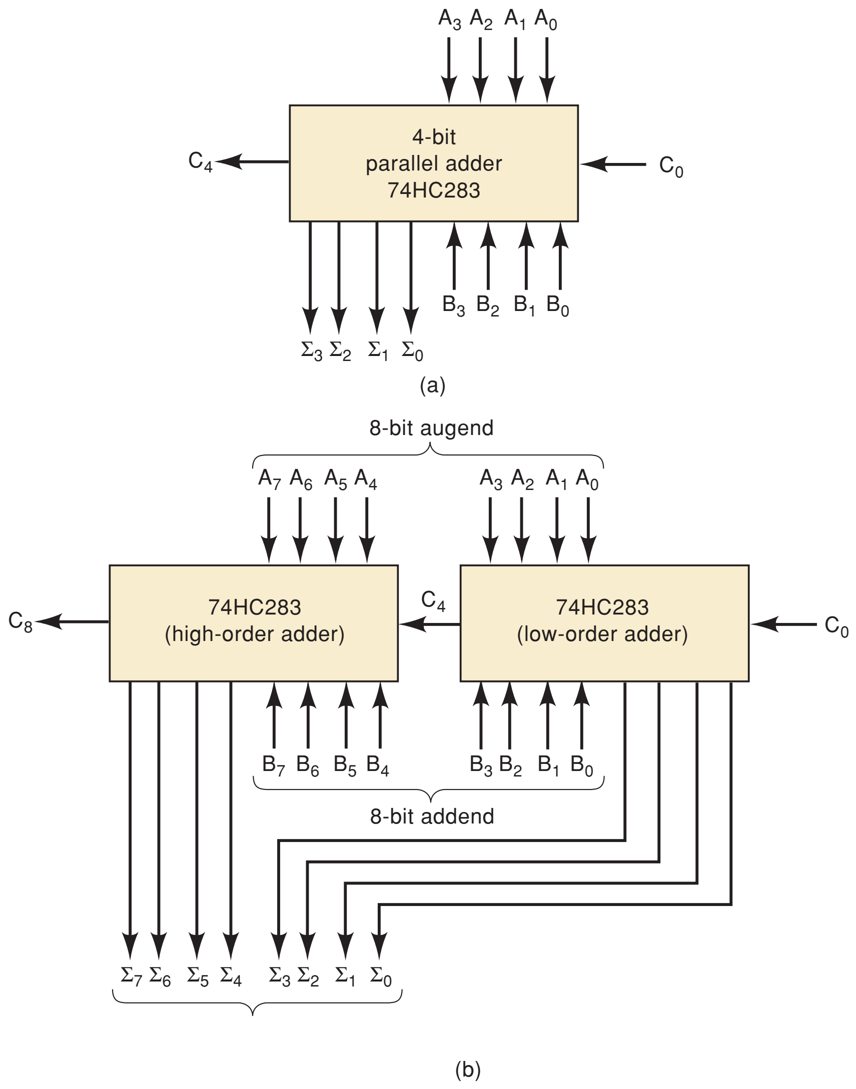
6-14 INTEGRATED-CIRCUIT PARALLEL ADDER

Several parallel adders are available as ICs. The most common is a four-bit parallel adder IC that contains four interconnected FAs and the look-ahead carry circuitry needed for high-speed operation. The 7483A, 74LS83A, 74LS283, and 74HC283 are all four-bit parallel-adder chips.

Figure 6-11(a) shows the functional symbol for the 74HC283 four-bit parallel adder (and its equivalents). The inputs to this IC are two four-bit numbers, $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$, and the carry, C_0, into the LSB position. The outputs are the sum bits and the carry, C_4, out of the MSB position. The sum bits are labeled $\Sigma_3\Sigma_2\Sigma_1\Sigma_0$, where Σ is the Greek capital letter *sigma*. The Σ label is just a common alternative to the S label for a sum bit.

Cascading Parallel Adders

Two or more IC adders can be connected together (cascaded) to accomplish the addition of larger binary numbers. Figure 6-11(b) shows two 74HC283 adders connected to add two 8-bit numbers A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0 and B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0. The adder on the right adds the lower-order bits of the numbers. The adder on the left adds the higher-order bits *plus* the C_4 carry out of the lower-order adder. The eight sum outputs are the resultant sum of the two



A_3 A_2 A_1 A_0
4-bit parallel adder 74HC283
C_4
C_0
B_3 B_2 B_1 B_0
Σ_3 Σ_2 Σ_1 Σ_0
(a)
8-bit augend
A_7 A_6 A_5 A_4
A_3 A_2 A_1 A_0
C_8
74HC283 (high-order adder)
C_4
74HC283 (low-order adder)
C_0
B_7 B_6 B_5 B_4
B_3 B_2 B_1 B_0
8-bit addend
Σ_7 Σ_6 Σ_5 Σ_4 Σ_3 Σ_2 Σ_1 Σ_0
(b)

FIGURE 6-11 (a) Block symbol for the 74HC283 four-bit parallel adder; (b) cascading two 74HC283s.

8-bit numbers. C_8 is the carry out of the MSB position. It can be used as the carry input to a third adder stage if larger binary numbers are to be added.

The look-ahead carry feature of the 74HC283 speeds up the operation of this two-stage adder because the logic level at C_4, the carry out of the lower-order stage, is generated more rapidly than it would be if there were no look-ahead carry circuitry on the 74HC283 chip. This allows the higher-order stage to produce its sum outputs more quickly.

EXAMPLE 6-10

Determine the logic levels at the inputs and outputs of the eight-bit adder in Figure 6-11(b) when 72_{10} is added to 137_{10}.

Solution

First, convert each number to an eight-bit binary number:

$$\begin{aligned} 137 &= 10001001 \\ 72 &= 01001000 \end{aligned}$$

These two binary values will be applied to the A and B inputs; that is, the A inputs will be 10001001 from left to right, and the B inputs will be 01001000 from left to right. The adder will produce the binary sum of the two numbers:

$$\begin{aligned}[A] &= 10001001\\ [B] &= \underline{01001000}\\ [\Sigma] &= 11010001\end{aligned}$$

The sum outputs will read 11010001 from left to right. There is no overflow into the C_8 bit, and so it will be a 0.

REVIEW QUESTIONS

1. How many 74HC283 chips are needed to add two 20-bit numbers?
2. If a 74HC283 has a maximum propagation delay of 30 ns from C_0 to C_4, what will be the total propagation delay of a 32-bit adder constructed from 74HC283s?
3. What will be the logic level at C_4 in Example 6-10?

6-15 2's-COMPLEMENT SYSTEM

Most modern computers use the 2's-complement system to represent negative numbers and to perform subtraction. The operations of addition and subtraction of signed numbers can be performed using only the addition operation if we use the 2's-complement form to represent negative numbers.

Addition

Positive and negative numbers, including the sign bits, can be added together in the basic parallel-adder circuit when the negative numbers are in 2's-complement form. This is illustrated in Figure 6-12 for the addition of -3 and $+6$. The -3 is represented in its 2's-complement form as 1101, where the first 1 is the sign bit; the $+6$ is represented as 0110, with the first zero as the sign bit. These numbers are stored in their corresponding registers. The four-bit parallel adder produces sum outputs of 0011, which represents $+3$. The C_4 output is 1, but remember that it is disregarded in the 2's-complement method.

Subtraction

When the 2's-complement system is used, the number to be subtracted (the subtrahend) is changed to its 2's complement and then *added* to the minuend (the number the subtrahend is being subtracted from). For example, we can assume that the minuend is already stored in the accumulator (A register). The subtrahend is then placed in the B register (in a computer it would be transferred here from memory) and is changed to its 2's-complement form before it is added to the number in the A register. The sum outputs of the adder circuit now represent the *difference* between the minuend and the subtrahend.

The parallel-adder circuit that we have been discussing can be adapted to perform the subtraction described above if we provide a means for taking

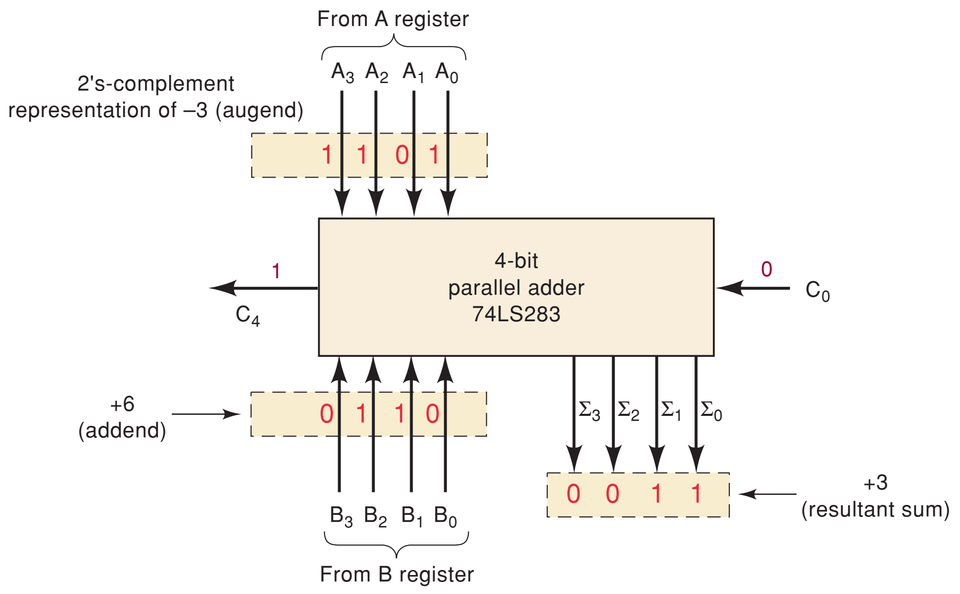


FIGURE 6-12 Parallel adder used to add and subtract numbers in 2's-complement system.

the 2's complement of the B register number. The 2's complement of a binary number is obtained by complementing (inverting) each bit and then adding 1 to the LSB. Figure 6-13 shows how this can be accomplished. The *inverted* outputs of the B register are used rather than the normal outputs; that is, $\overline{B}_0, \overline{B}_1, \overline{B}_2$, and $\overline{B}_3$ are fed to the adder inputs (remember, B_3 is the sign bit). This takes care of complementing each bit of the B number. Also, C_0 is made a logical 1, so that it adds an extra 1 into the LSB of the adder; this accomplishes the same effect as adding 1 to the LSB of the B register for forming the 2's complement.

The outputs Σ_3 to Σ_0 represent the results of the subtraction operation. Of course, Σ_3 is the sign bit of the result and indicates whether the result is $+$ or $-$. The carry output C_4 is again disregarded.

To help clarify this operation, study the following steps for subtracting +6 from +4:

1. +4 is stored in the A register as 0100.
2. +6 is stored in the B register as 0110.
3. The inverted outputs of the B-register FFs (1001) are fed to the adder.

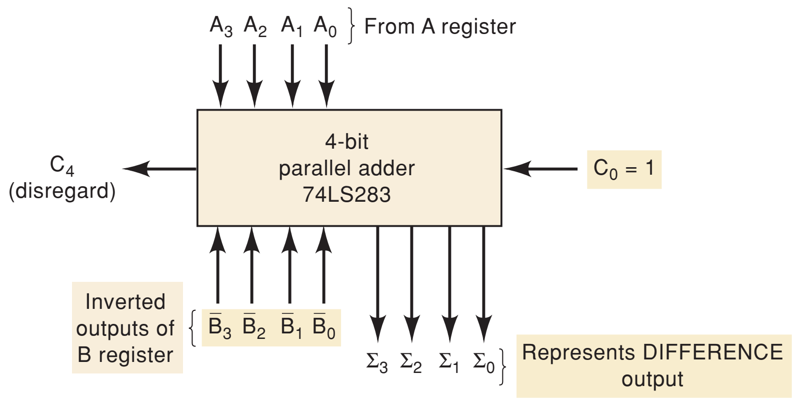

A_3 A_2 A_1 A_0
From A register
C_4 (disregard)
4-bit parallel adder 74LS283
$C_0 = 1$
Inverted outputs of B register
$\overline{B}_3$ $\overline{B}_2$ $\overline{B}_1$ $\overline{B}_0$
Σ_3 Σ_2 Σ_1 Σ_0
Represents DIFFERENCE output

FIGURE 6-13 Parallel adder used to perform subtraction ($A - B$) using the 2's-complement system. The bits of the subtrahend (B) are inverted, and $C_0 = 1$ to produce the 2's complement.

4. The parallel-adder circuitry adds $[A] = 0100$ to $[\overline{B}] = 1001$ along with a carry, $C_0 = 1$, into the LSB. The operation is shown below.

$$\begin{array}{rl} 1 & \leftarrow C_0 \\ 0100 & \leftarrow [A] \\ +\ \underline{1001} & \leftarrow [\,\overline{B}\,] \\ 1110 & \leftarrow [\Sigma] = [A] - [B] \end{array}$$

The result at the sum outputs is 1110. This actually represents the result of the *subtraction* operation, the *difference* between the number in the A register and the number in the B register, that is, $[A] - [B]$. Because the sign bit $= 1$, it is a negative result and is in 2's-complement form. We can verify that 1110 represents -2_{10} by taking its 2's complement and obtaining $+2_{10}$:

$$\begin{array}{rl} 1110 & \\ 0001 & \\ +\ \underline{\quad 1} & \\ 0010 & = +2_{10} \end{array}$$

Combined Addition and Subtraction

It should now be clear that the basic parallel-adder circuit can be used to perform addition or subtraction depending on whether the B number is left unchanged or is converted to its 2's complement. A complete circuit that can perform *both* addition and subtraction in the 2's-complement system is shown in Figure 6-14.

This **adder/subtractor** circuit is controlled by the two control signals ADD and SUB. When the ADD level is HIGH, the circuit performs addition of the numbers stored in the A and B registers. When the SUB level is HIGH, the circuit subtracts the B-register number from the A-register number. The operation is described as follows:

1. Assume that ADD = 1 and SUB = 0. The SUB = 0 *disables* (inhibits) AND gates 2, 4, 6, and 8, holding their outputs at 0. The ADD = 1 *enables* AND gates 1, 3, 5, and 7, allowing their outputs to pass the B_0, B_1, B_2, and B_3 levels, respectively.
2. The levels B_0 to B_3 pass through the OR gates into the four-bit parallel adder to be added to the bits A_0 to A_3. The *sum* appears at the outputs Σ_0 to Σ_3.
3. Note that SUB = 0 causes a carry $C_0 = 0$ into the adder.
4. Now assume that ADD = 0 and SUB = 1. The ADD = 0 inhibits AND gates 1, 3, 5, and 7. The SUB = 1 enables AND gates 2, 4, 6, and 8, so that their outputs pass the $\overline{B}_0$, $\overline{B}_1$, $\overline{B}_2$, and $\overline{B}_3$ levels, respectively.
5. The levels $\overline{B}_0$ to $\overline{B}_3$ pass through the OR gates into the adder to be added to the bits A_0 to A_3. Note also that C_0 is now 1. Thus, the B-register number has essentially been converted to its 2's complement.
6. The *difference* appears at the *outputs* Σ_0 to Σ_3.

Circuits like the adder/subtractor of Figure 6-14 are used in computers because they provide a relatively simple means for adding and subtracting signed binary numbers. In most computers, the outputs present at the Σ output lines are usually transferred into the A register (accumulator), so that the results of the addition or subtraction always end up stored in the A register. This is accomplished by applying a TRANSFER pulse to the *CLK* inputs of register A.

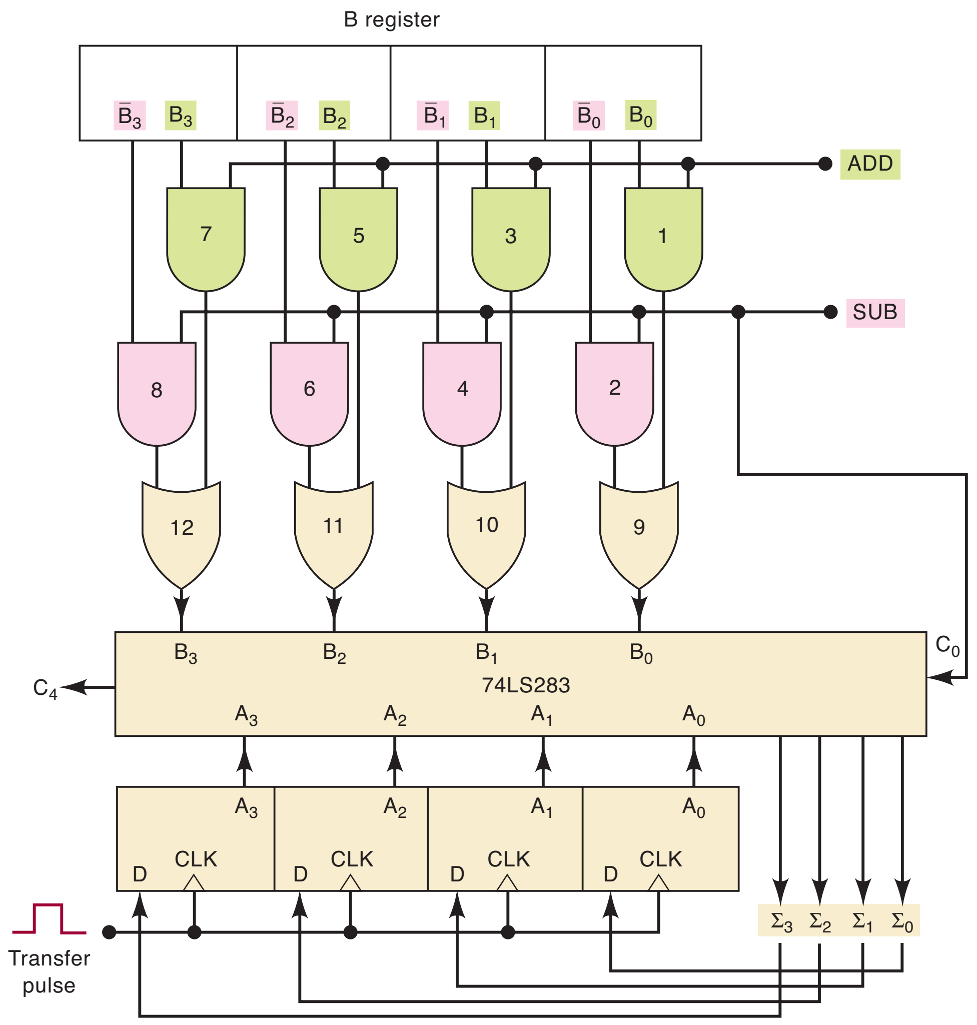

B register
$\bar{B}_3$ B_3 $\bar{B}_2$ B_2 $\bar{B}_1$ B_1 $\bar{B}_0$ B_0
ADD
SUB
7
5
3
1
8
6
4
2
12
11
10
9
B_3 B_2 B_1 B_0
C_0
C_4
74LS283
A_3 A_2 A_1 A_0
D CLK
Σ_3 Σ_2 Σ_1 Σ_0
Transfer pulse

FIGURE 6-14 Parallel adder/subtractor using the 2's-complement system.

REVIEW QUESTIONS

1. Why does C_0 have to be a 1 in order to use the adder circuit in Figure 6-13 as a subtractor?
2. Assume that $[A] = 0011$ and $[B] = 0010$ in Figure 6-14. If ADD $= 1$ and SUB $= 0$, determine the logic levels at the OR gate outputs.
3. Repeat question 2 for ADD $= 0$, SUB $= 1$.
4. *True or false:* When the adder/subtractor circuit is used for subtraction, the 2's complement of the subtrahend appears at the input of the adder.

6-16 ALU INTEGRATED CIRCUITS

Several integrated circuits are called arithmetic/logic units (ALUs), even though they do not have the full capabilities of a computer's arithmetic/logic unit. These ALU chips are capable of performing several different arithmetic and logic operations on binary data inputs. The specific operation that an ALU IC is to perform is determined by a specific binary code applied to its function-select inputs. Some of the ALU ICs are fairly complex, and it would require a great amount of time and space to explain and illustrate their operation. In this section, we will use a relatively simple, yet useful, ALU chip to show the basic concepts behind all ALU chips. The ideas presented here can then be extended to the more complex devices.

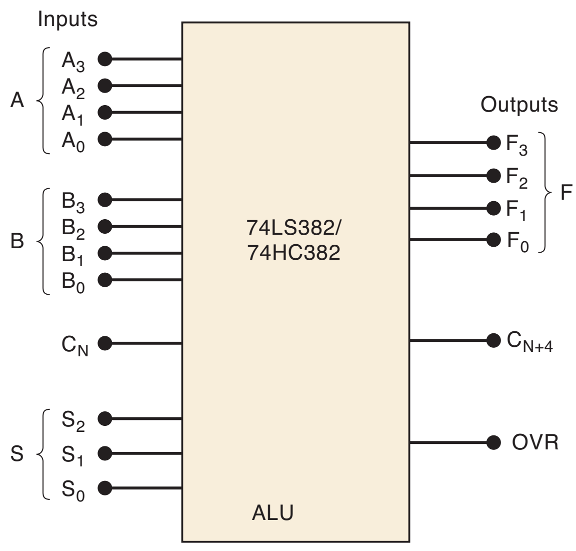
The 74LS382/HC382 ALU

Figure 6-15(a) shows the block symbol for an ALU that is available as a 74LS382 (TTL) and as a 74HC382 (CMOS). This 20-pin IC operates on two four-bit input numbers, $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$, to produce a four-bit output result $F_3F_2F_1F_0$. This ALU can perform *eight* different operations. At any given time, the operation that it is performing depends on the input code applied to the function-select inputs $S_2S_1S_0$. The table in Figure 6-15(b) shows the eight available operations. We will now describe each of these operations.

CLEAR OPERATION With $S_2S_1S_0 = 000$, the ALU will *clear* all of the bits of the F output so that $F_3F_2F_1F_0 = 0000$.

ADD OPERATION With $S_2S_1S_0 = 011$, the ALU will add $A_3A_2A_1A_0$ to $B_3B_2B_1B_0$ to produce their sum at $F_3F_2F_1F_0$. For this operation, C_N is the carry into the LSB position, and it must be made a 0. C_{N+4} is the carry output from the MSB position. *OVR* is the overflow indicator output; it detects overflow when signed numbers are being used. *OVR* will be a 1 when an add or a subtract operation produces a result that is too large to fit into four bits (including the sign bit).

SUBTRACT OPERATIONS With $S_2S_1S_0 = 001$, the ALU will subtract the A input number from the B input number. With $S_2S_1S_0 = 010$, the ALU will subtract B from A. In either case, the difference appears at $F_3F_2F_1F_0$. Note that the subtract operations require that the C_N input be a 1.



A = 4-bit input number
B = 4-bit input number
C_N = carry into LSB position
S = 3-bit operation select inputs

F = 4-bit output number
C_{N+4} = carry out of MSB position
OVR = overflow indicator

(a)

Function Table

S_2	S_1	S_0	Operation	Comments
0	0	0	CLEAR	$F_3F_2F_1F_0 = 0000$
0	0	1	B minus A	Needs $C_N = 1$
0	1	0	A minus B	
0	1	1	A plus B	Needs $C_N = 0$
1	0	0	$A \oplus B$	Exclusive-OR
1	0	1	A + B	OR
1	1	0	AB	AND
1	1	1	PRESET	$F_3F_2F_1F_0 = 1111$

Notes: S inputs select operation.
OVR = 1 for signed-number overflow.

(b)

FIGURE 6-15 (a) Block symbol for 74LS382/HC382 ALU chip; (b) function table showing how select inputs (S) determine what operation is to be performed on A and B inputs.

XOR OPERATION With $S_2S_1S_0 = 100$, the ALU will perform a bit-by-bit XOR operation on the A and B inputs. This is illustrated below for $A_3A_2A_1A_0 = 0110$ and $B_3B_2B_1B_0 = 1100$.

$$\begin{aligned} A_3 \oplus B_3 &= 0 \oplus 1 = 1 = F_3 \\ A_2 \oplus B_2 &= 1 \oplus 1 = 0 = F_2 \\ A_1 \oplus B_1 &= 1 \oplus 0 = 1 = F_1 \\ A_0 \oplus B_0 &= 0 \oplus 0 = 0 = F_0 \end{aligned}$$

The result is $F_3F_2F_1F_0 = 1010$.

OR OPERATION With $S_2S_1S_0 = 101$, the ALU will perform a bit-by-bit OR operation on the A and B inputs. For example, with $A_3A_2A_1A_0 = 0110$ and $B_3B_2B_1B_0 = 1100$, the ALU will generate a result of $F_3F_2F_1F_0 = 1110$.

AND OPERATION With $S_2S_1S_0 = 110$, the ALU will perform a bit-by-bit AND operation on the A and B inputs. For example, with $A_3A_2A_1A_0 = 0110$ and $B_3B_2B_1B_0 = 1100$, the ALU will generate a result of $F_3F_2F_1F_0 = 0100$.

PRESET OPERATIONS With $S_2S_1S_0 = 111$, the ALU will *set* all of the bits of the output so that $F_3F_2F_1F_0 = 1111$.

EXAMPLE 6-11

(a) Determine the 74HC382 outputs for the following inputs: $S_2S_1S_0 = 010$, $A_3A_2A_1A_0 = 0100$, $B_3B_2B_1B_0 = 0001$, and $C_N = 1$.

(b) Change the select code to 011 and repeat.

Solution

(a) From the function table in Figure 6-15(b), 010 selects the $(A - B)$ operation. The ALU will perform the 2's-complement subtraction by complementing B and adding it to A and C_N. Note that $C_N = 1$ is needed to complete the 2's complement of B effectively.

$$\begin{array}{rrl} & 1 & \leftarrow C_N \\ & 0100 & \leftarrow A \\ + & \underline{1110} & \leftarrow \overline{B} \\ & 10011 & \end{array}$$

$$C_{N+4} \longrightarrow\!\uparrow \quad \uparrow\!\longleftarrow F_3\,F_2\,F_1\,F_0$$

As always in 2's-complement subtraction, the CARRY OUT of the MSB is discarded. The correct result of the $(A - B)$ operation appears at the F outputs.

The OVR output is determined by considering the input numbers to be signed numbers. Thus, we have $A_3A_2A_1A_0 = 0100 = +4_{10}$ and $B_3B_2B_1B_0 = 0001 = +1_{10}$. The result of the subtract operation is$F_3F_2F_1F_0 = 0011 = +3_{10}$, which is correct. Therefore, no overflow has occurred, and $OVR = 0$. If the result had been negative, it would have been in 2's-complement form.

(b) A select code of 011 will produce the sum of the A and B inputs. However, because $C_N = 1$, there will be a carry of 1 added into the LSB position. This will produce a result of $F_3F_2F_1F_0 = 0110$, which is 1 greater than $(A + B)$. The C_{N+4} and OVR outputs will both be 0. For the correct sum to appear at F, the C_N input must be at 0.

Expanding the ALU

A single 74LS382 or 74HC382 operates on four-bit numbers. Two or more of these chips can be connected together to operate on larger numbers. Figure 6-16 shows how two four-bit ALUs can be combined to add two eight-bit numbers, $B_7B_6B_5B_4B_3B_2B_1B_0$ and $A_7A_6A_5A_4A_3A_2A_1A_0$, to produce the output sum $\Sigma_7\Sigma_6\Sigma_5\Sigma_4\Sigma_3\Sigma_2\Sigma_1\Sigma_0$. Study the circuit diagram and note the following points:

1. Chip Z1 operates on the four lower-order bits of the two input numbers. Chip Z2 operates on the four higher-order bits.
2. The sum appears at the *F* outputs of Z1 and Z2. The lower-order bits appear at Z1, and the higher-order bits appear at Z2.
3. The C_N input of Z1 is the carry into the LSB position. For addition, it is made a 0.
4. The carry output $[C_{N+4}]$ of Z1 is connected to the carry input $[C_N]$ of Z2.
5. The *OVR* output of Z2 is the overflow indicator when signed eight-bit numbers are being used.
6. The corresponding select inputs of the two chips are connected together so that Z1 and Z2 are always performing the same operation. For addition, the select inputs are shown as 011.

EXAMPLE 6-12

How would the arrangement of Figure 6-16 have to be changed in order to perform the subtraction $(B - A)$?

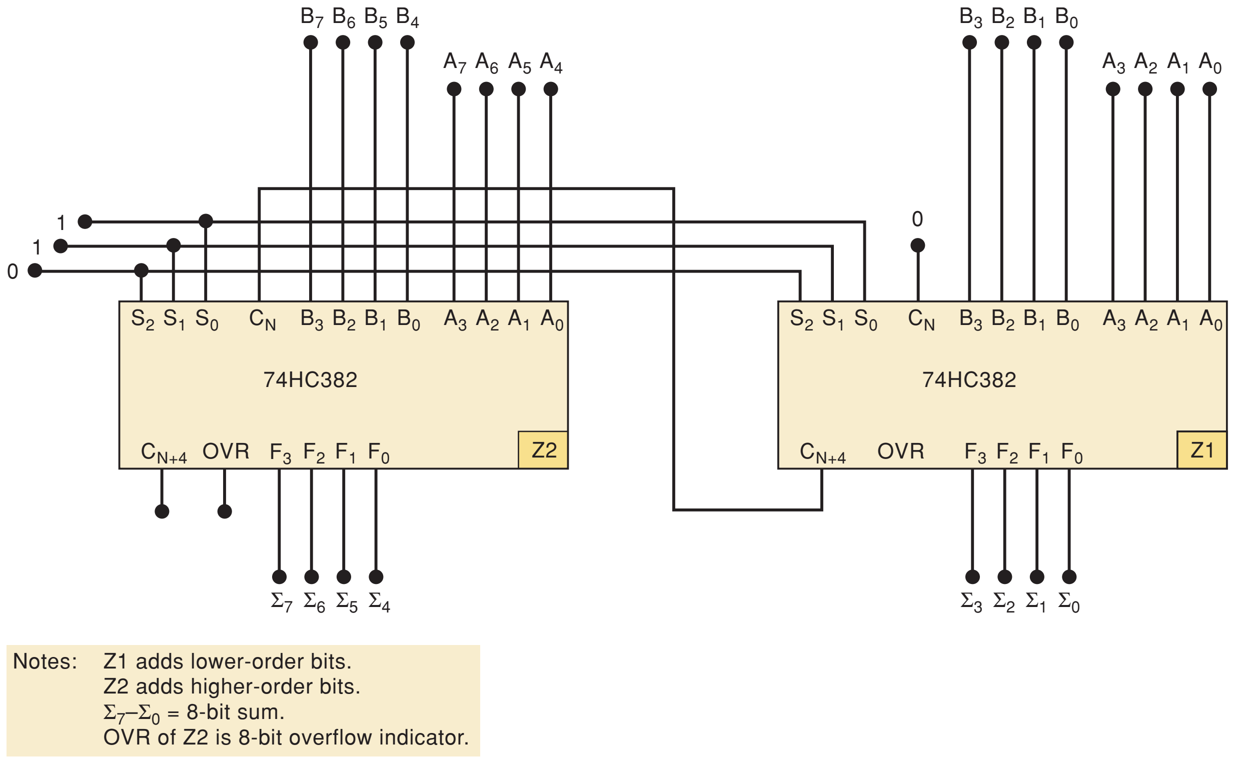

B_7 B_6 B_5 B_4
A_7 A_6 A_5 A_4
B_3 B_2 B_1 B_0
A_3 A_2 A_1 A_0
1
1
0
0
S_2 S_1 S_0 C_N B_3 B_2 B_1 B_0 A_3 A_2 A_1 A_0
74HC382
C_{N+4} OVR F_3 F_2 F_1 F_0
Z2
Z1
Σ_7 Σ_6 Σ_5 Σ_4
Σ_3 Σ_2 Σ_1 Σ_0
Notes: Z1 adds lower-order bits.
Z2 adds higher-order bits.
Σ_7–Σ_0 = 8-bit sum.
OVR of Z2 is 8-bit overflow indicator.

FIGURE 6-16 Two 74HC382 ALU chips connected as an eight-bit adder.

Solution

The select input code [see the table in Figure 6-15(b)] must be changed to 001, and the C_N input of Z1 must be made a 1.

Other ALUs

The 74LS181/HC181 is another four-bit ALU. It has four select inputs that can select any of 16 different operations. It also has a mode input bit that can switch between logic operations and arithmetic operations (add and subtract). This ALU has an $A = B$ output that is used to compare the magnitudes of the A and B inputs. When the two input numbers are exactly equal, the $A = B$ output will be a 1; otherwise, it is a 0.

The 74LS881/HC881 is similar to the 181 chip, but it has the capability of performing some additional logic operations.

REVIEW QUESTIONS

1. Apply the following inputs to the ALU of Figure 6-15, and determine the outputs: $S_2S_1S_0 = 001$, $A_3A_2A_1A_0 = 1110$, $B_3B_2B_1B_0 = 1001$, $C_N = 1$.
2. Change the select code to 011 and C_N to 0, and repeat review question 1.
3. Change the select code to 110, and repeat review question 1.
4. Apply the following inputs to the circuit of Figure 6-16, and determine the outputs: $B = 01010011$, $A = 00011000$.
5. Change the select code to 111, and repeat review question 4.
6. How many 74HC382s are needed to add two 32-bit numbers?

6-17 TROUBLESHOOTING CASE STUDY

A technician is testing the adder/subtractor redrawn in Figure 6-17 and records the following test results for the various operating modes:

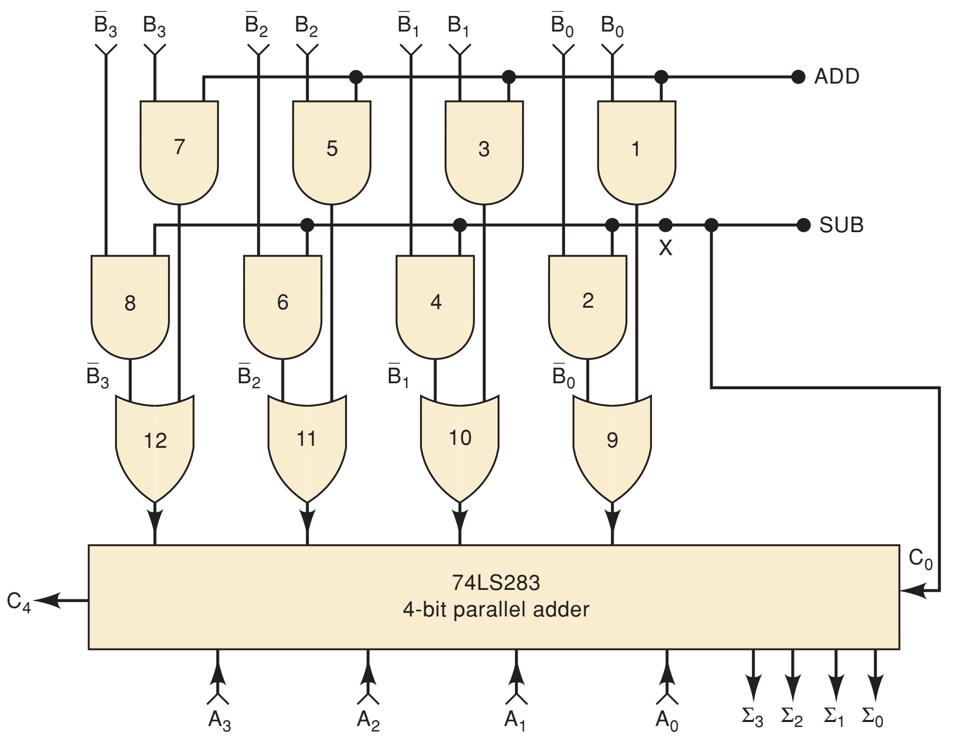
Mode 1: ADD = 0, SUB = 0. The sum outputs are always equal to the number in the A register *plus one.* For example, when $[A] = 0110$, the sum is $[\Sigma] = 0111$. This is incorrect because the OR outputs and C_0 should all be 0 in this mode to produce $[\Sigma] = [A]$.

Mode 2: Add = 1, SUB = 0. The sum is always 1 more than it should be. For example, with $[A] = 0010$ and $[B] = 0100$, the sum output is 0111 instead of 0110.

Mode 3: Add = 0, SUB = 1. The Σ outputs are always equal to $[A] - [B]$, as expected.

When she examines these test results, the technician sees that the sum outputs exceed the expected results by 1 for the first two modes of operation. At first, she suspects a possible fault in one of the LSB inputs to the adder, but she dismisses this because such a fault would also affect the subtraction operation, which is working correctly. Eventually, she realizes that there is another fault that could add an extra 1 to the results for the first two modes without causing an error in the subtraction mode.

Recall that C_0 is made a 1 in the subtraction mode as part of the 2's-complement operation on $[B]$. For the other modes, C_0 is to be a 0. The technician



$\overline{B}_3$ B_3 $\overline{B}_2$ B_2 $\overline{B}_1$ B_1 $\overline{B}_0$ B_0
ADD
SUB
X
7
5
3
1
8
6
4
2
12
11
10
9
74LS283
4-bit parallel adder
C_4
C_0
A_3 A_2 A_1 A_0
Σ_3 Σ_2 Σ_1 Σ_0

FIGURE 6-17 Parallel adder/subtractor circuit.

checks the connection between the *SUB* signal and the C_0 input to the adder and finds that it is open due to a bad solder connection. This open connection explains the observed results because the TTL adder responds as if C_0 were a constant logic 1, causing an extra 1 to be added to the result in modes 1 and 2. The open connection would have no effect on mode 3 because C_0 is supposed to be a 1 anyway.

EXAMPLE 6-13

Consider again the adder/subtractor circuit. Suppose that there is a break in the connection path between the SUB input and the AND gates at point X in Figure 6-17. Describe the effects of this open on the circuit operation for each mode.

Solution

First, realize that this fault will produce a logic 1 at the affected input of AND gates 2, 4, 6, and 8, which will permanently enable each of these gates to pass its $\overline{B}$ input to the following OR gate as shown.

Mode 1: ADD = 0, SUB = 0. The fault will cause the circuit to perform subtraction—almost. The 1's complement of $[B]$ will reach the OR gate outputs and be applied to the adder along with $[A]$. With $C_0 = 0$, the 2's complement of $[B]$ will not be complete; it will be short by 1. Thus, the adder will produce $[A] - [B] - 1$. To illustrate, let's try $[A] = +6 = 0110$ and $[B] = +3 = 0011$. The adder will add as follows:

$$\begin{aligned} \text{1's complement of } [B] &= \phantom{\cancel{1}}1100 \\ [A] &= \phantom{\cancel{1}}0110 \\ \text{result} &= \cancel{1}0010 \end{aligned}$$

Disregard carry.

The result is 0010 = +2 instead of 0011 = +3, as it would be for normal subtraction.

Mode 2: ADD = 1, SUB = 0. With ADD = 1, AND gates 1, 3, 5, and 7 will pass the B inputs to the following OR gate. Thus, each OR gate will have a $\overline{B}$ and a B at its inputs, thereby producing a 1 output. For example, the inputs to OR gate 9 will be $\overline{B}_0$ coming from AND gate 2 (because of the fault), and B_0 coming from AND gate 1 (because ADD = 1). Thus, OR gate 9 will produce an output of $\overline{B}_0 + B_0$, which will always be a logic 1.

The adder will add the 1111 from the OR gates to the $[A]$ to produce a sum that is 1 less than $[A]$. Why? Because $1111_2 = -1_{10}$.

Mode 3: ADD = 0, SUB = 1. This mode will work correctly because SUB = 1 is supposed to enable AND gates 2, 4, 6, and 8 anyway.

6-18 USING TTL LIBRARY FUNCTIONS WITH ALTERA

The adder and ALU ICs that we have looked at in this chapter are just a few of the many MSI chips that have served as the building blocks of digital systems for decades. Whenever a technology serves such a long and useful lifetime, it has a lasting impact on the field and the people who use it. TTL integrated circuits certainly fall into this category and continue in various forms today. Experienced engineers and technicians (we want to avoid the word *old*) are familiar with the standard parts. Existing designs can be remanufactured and upgraded using the same basic circuits if they can be implemented in a VLSI PLD. Data sheets for these devices are readily available, and studying these old TTL parts is still an excellent way to learn the fundamentals of any digital system.

For all of these reasons, the Altera development system offers what they refer to as old-style macrofunctions. A **macrofunction** is a self-contained description of a logic circuit with all its inputs, outputs, and operational characteristics defined. In other words, they have gone to the trouble of writing the code necessary to get a PLD to emulate the operation of many conventional TTL MSI devices. All the designer needs to know is how to hook it into the rest of the system. In this section, we will expand on the concepts of logic primitives and libraries presented in Chapter 5 to see how we can use standard MSI parts in our designs.

The 74382 arithmetic logic unit (ALU) is a fairly sophisticated IC. The task of describing its operation using HDL code is challenging but certainly within our reach. Refer again to the examples of this IC and its operation, which were covered in Section 6-16. Specifically, look at Figure 6-16, which shows how to cascade two four-bit ALU chips to make an eight-bit ALU that could serve as the heart of a microcontroller's central processing unit (CPU). Figure 6-18 shows the graphic method of describing the eight-bit circuit using Altera's graphic description file and macrofunction blocks from its library of components. The 74382 symbols are simply chosen from the list in the macrofunction library and placed on the screen. Wiring these chips together is simple and intuitive.

It is possible to connect standard library MSI parts using only HDL. Just as we demonstrated connecting flip-flop primitives together to make more complex circuits, we can connect ICs such as the 74382 to other parts. The names of all the input and output ports on these standard parts are defined

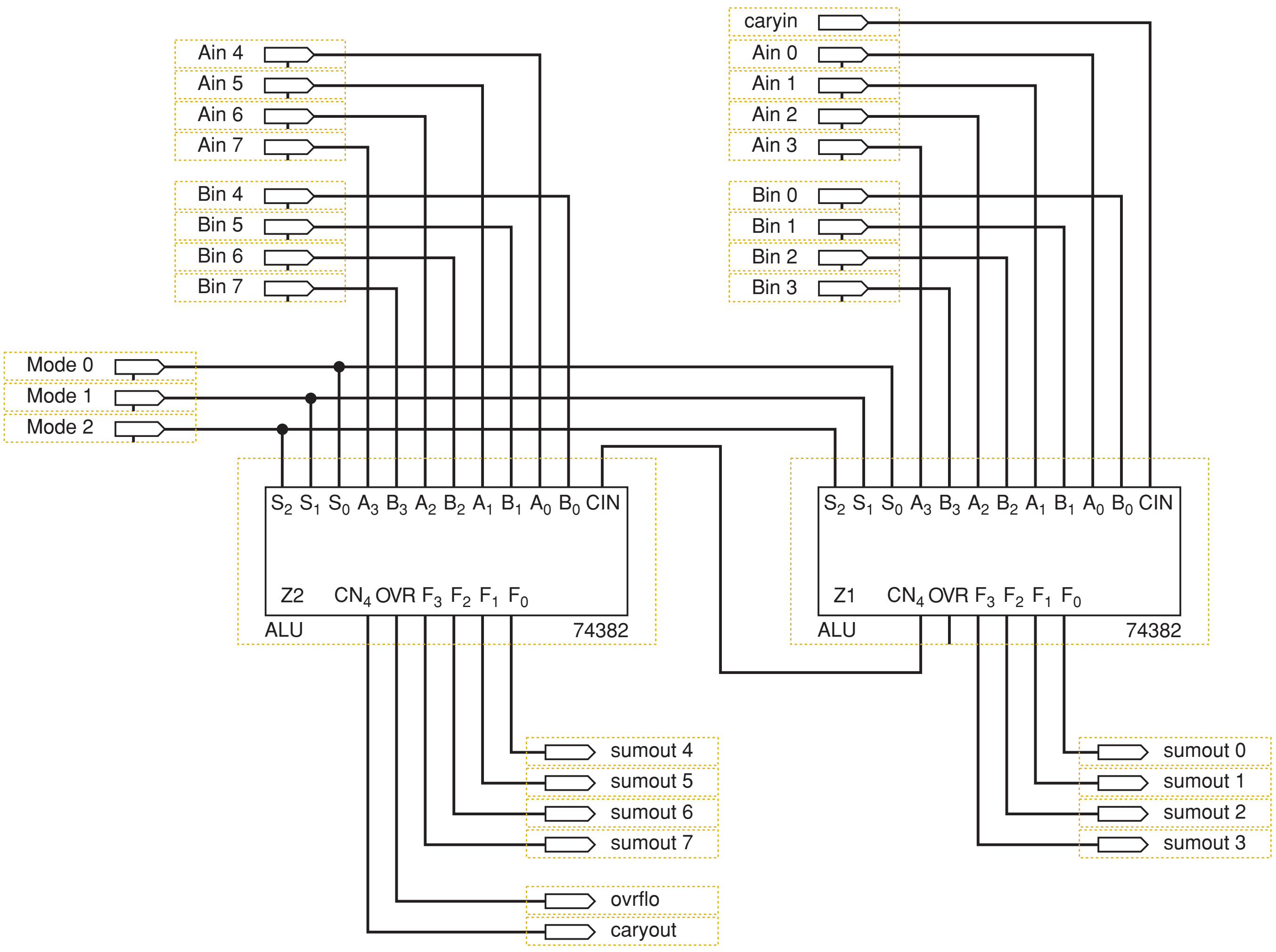


FIGURE 6-18 An Altera graphic description file of an eight-bit ALU.

in a **function prototype**, which can be found in the help menu. The function prototype given for a 74382 is

AHDL Function Prototype (port name and order also apply to Verilog HDL):

```
FUNCTION 74382 (s[2..0], a[3..0], b[3..0], cin)
RETURNS (ovr, cn4, f[3..0]);
```

REVIEW QUESTIONS

1. Where can you find information about using a 74283 full adder in your HDL design?
2. What is a macrofunction?

6-19 LOGICAL OPERATIONS ON BIT ARRAYS

In the previous section, we examined the use of macrofunctions to build systems from standard parts. Now we need to practice writing HDL code rather than using a macrofunction to make an adder similar to Figure 6-6. In this section, we will expand our understanding of the HDL techniques in two main areas: specifying groups of bits in an array and using logical operations to combine arrays of bits using Boolean expressions.

In Section 6-12, we discussed register notation, which makes it easy to describe the contents of registers and signals consisting of multiple bits. The HDLs use arrays of bits in a similar notation to describe signals, as we discussed in Chapter 4. For example, in AHDL, the four-bit signal named *d* is defined as:

```
VARIABLE d[3..0] :NODE.
```

In VHDL, the same data format is expressed as:

```
SIGNAL d :BIT_VECTOR (3 DOWNTO 0).
```

Each bit in these data types is designated by an element number. In this example of a bit array named *d*, the bits can be referred to as d3, d2, d1, d0. Bits can also be grouped into **sets**. For example, if we want to refer to the three most significant bits of *d* as a set, we can use the expression d[3..1] in AHDL and the expression d (3 DOWNTO 1) in VHDL. Once a value is assigned to the array and the desired set of bits is identified, logical operations can be performed on the entire set of bits. As long as the sets are the same size (the same number of bits), two sets can be combined in a logical expression, just like you would combine single variables in a Boolean equation. Each of the corresponding pairs of bits in the two sets is combined as stated in the logic equation. This allows one equation to describe the logical operation performed on each bit in a set.

EXAMPLE 6-14

Assume D_3, D_2, D_1, D_0 has the value 1011 and G_3, G_2, G_1, G_0 has the value 1100. Let's define Dnum = $[D_3, D_2, D_1, D_0]$ and Gnum = $[G_3, G_2, G_1, G_0]$. Let's also define Y = $[Y_3, Y_2, Y_1, Y_0]$ where Y is related to Dnum and Gnum as follows:

Y = Dnum • Gnum;

What is the value of X after this operation?

Solution

$$\begin{array}{llll} D_3, D_2, D_1, D_0 & 1\ 0\ 1\ 1 & \\ \updownarrow\ \ \updownarrow\ \ \updownarrow\ \ \updownarrow & \updownarrow\updownarrow\updownarrow\updownarrow & \text{AND each bit position together} \\ G_3, G_2, G_1, G_0 & 1\ 1\ 0\ 0 & \\ \hline \mathrm{Y}_3, \mathrm{Y}_2, \mathrm{Y}_1, \mathrm{Y}_0 & 1\ 0\ 0\ 0 & \end{array}$$

Thus, Y is a set of four bits with value 1000.

EXAMPLE 6-15

For the register values described in Example 6-14, declare each *d*, *g*, and *x*. Then write an expression using your favorite HDL that performs the ANDing operation on all bits.

Solution

```
SUBDESIGN bitwise_and
(    d[3..0], g[3..0]       :INPUT;
     y[3..0]                :OUTPUT;)
BEGIN
     y[] = d[] & g[];
END;
```

```
ENTITY bitwise_and IS
PORT(d, g           :IN BIT_VECTOR (3 DOWNTO 0);
     y              :OUT BIT_VECTOR (3 DOWNTO 0));
END bitwise_and;
ARCHITECTURE a OF bitwise_and IS
BEGIN
     y <= d AND g;
END a;
```

6-20 HDL ADDERS

In this section, we will see how to create a parallel adder circuit that can be used to add bit arrays using the logic equation for a single-bit full adder. Figure 6-19 shows the basic block diagram with signals labeled to create a four-bit adder. Notice that each bit of the augend [A], addend [B], carry [C], and sum [S] are bit array variables and have index numbers associated with them. The equation for the sum using register notation is:

$$[S] = [A] \oplus [B] \oplus [C];$$

Notice that the carry signals between stages are not inputs or outputs of this overall circuit, but rather intermediate variables. A strategy must be developed for labeling the carry bits so that they can be used in an array. We have chosen to let each carry bit serve as an input to its corresponding adder stage, as shown in Figure 6-19. For example, C0 is an input to the bit 0 stage, C1 is the carry input to the bit 1 stage, and so on. The bits of the carry array can be thought of as the "wires" that connect the adders. This array must have a carry in for each stage and also a carry out for the most significant state. In this example, there will be five bits, labeled C4 through C0, in the carry array. The set of data that represents the carry outputs would be C4 through C1, and the set of data that represents the carry inputs would be C3 through C0.

$$\begin{aligned} \text{Cin} &= \text{C}[3..0] \\ \text{Cout} &= \text{C}[4..1] \end{aligned}$$

The HDLs allow us to specify which sets of bits out of the entire array we want to use in an equation. To ensure that all variables being combined in a logic equation contain the same number of bits, we can start with the general equation for the carry out of a one-bit adder as follows:

$$\text{Cout} = \text{AB} + \text{ACin} + \text{BCin}$$

AUGEND	A_3	A_2	A_1	A_0	A
ADDEND	B_3	B_2	B_1	B_0	B
CARRYin	C_3	C_2	C_1	C_0	Cin
SUM	S_3	S_2	S_1	S_0	S

Generate sum
S = A ⊕ [B ⊕ Cin]

AUGEND	A_3	A_2	A_1	A_0	A
ADDEND	B_3	B_2	B_1	B_0	B
CARRYin	C_3	C_2	C_1	C_0	Cin
CARRYout	C_4	C_3	C_2	C_1	Cout

Generate carry bits
Cout = A•B + A•Cin + B•Cin

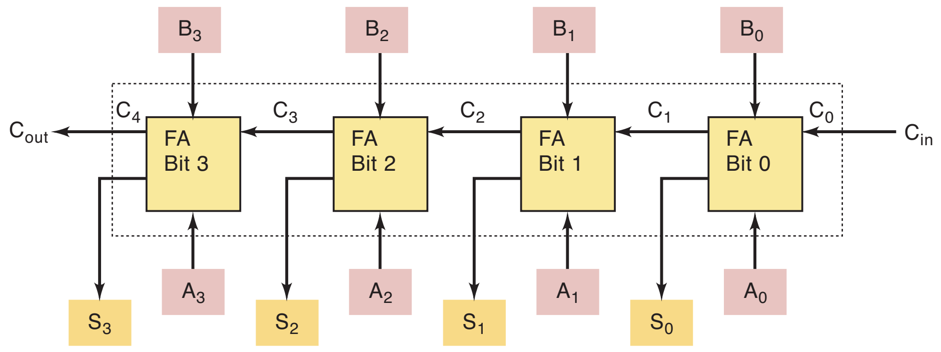


FIGURE 6-19 Four-bit parallel adder.

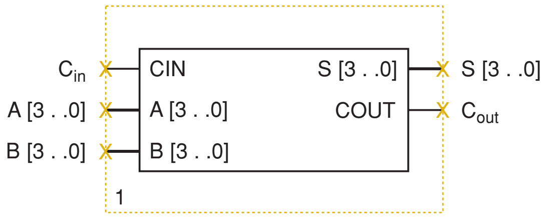
Substituting our definition of Cin and Cout (above), we have the following equation for the four-bit adder carry out:

C[4..1] = A[3..0] & B[3..0] + A[3..0] & C[3..0] + B[3..0] & C[3..0]

The graphic symbol for this device is shown in Figure 6-20. Notice that it does not show the carry array. It is a variable or signal *inside* the block. Altera allows any SUBDESIGN in AHDL or ENTITY in VHDL, even those that you create, to be represented by a graphic block diagram symbol such as this. This is all part of the hierarchical design scheme of the Altera development system (described in Chapter 4).

To summarize this information, Figure 6-19 shows the "insides" of the block diagram in Figure 6-20 and summarizes the operation described by the two equations. Now let's look at text-based files that can be used to generate a block symbol like the one in Figure 6-20 using AHDL and VHDL.

FIGURE 6-20 Block symbol generated by Altera MAX+PLUS.



AHDL FOUR-BIT ADDER

In lines 14 and 15 of the AHDL code of Figure 6-21, notice the syntax for referring to bit arrays in their entirety. The name is given, followed by [ ]. If no bits are designated inside the square brackets, it means that all the bits that were declared are included in the operations. Lines 14 and 15 describe fully all four adder circuits and come up with the sum. In order to choose a specific set of elements from the array (i.e., a subset of the array), the name is followed by the range of element numbers in square brackets. For example, the carry equation (line 15) in AHDL syntax is:

c[4..1] = a[] & b[] # a[] & c[3..0] # b[] & c[3..0];

Notice that only four bits of the carry array *c[ ]* are being assigned, even though the array is five bits long. In this way, the carry out of each single-bit adder is assigned as the carry in of the next stage.

```
 1  SUBDESIGN fig6_21
 2  (
 3        cin          :INPUT;      -- carry in
 4        a[3..0]      :INPUT;      -- augend
 5        b[3..0]      :INPUT;      -- addend
 6        s[3..0]      :OUTPUT;     -- sum
 7        cout         :OUTPUT;     -- carry OUT
 8  )
 9  VARIABLE
10        c[4..0]      :NODE;       -- carry array is 5 bits long!
11
12  BEGIN
13        c[0] = cin;
14        s[] = a[] $ b[] $ c[3..0];   -- generate sum
15        c[4..1] = (a[] & b[]) # (a[] & c[3..0]) # (b[] & c[3..0]);
16        cout = c[4];                 -- carry out
17  END;
```

FIGURE 6-21 AHDL adder.

VHDL FOUR-BIT ADDER

In the VHDL code of Figure 6-22, notice the syntax for referring to bit arrays in their entirety. The name is simply used with no bit designations. Lines 15 and 16 describe fully all four adder circuits that will come up with the sum. In order to choose a specific set of elements from the array (i.e., a subset of the array), the name is followed by the range of element numbers in parentheses. The carry equation (line 16) in VHDL syntax is:

c(4 DOWNTO 1) <= (a AND b) OR (a AND c(3 DOWNTO 0)) OR (b AND c(3 DOWNTO 0));

Notice that only four bits of the carry array *c* are being assigned, even though the array is five bits long. In this way, the carry out of each single-bit adder is assigned as the carry in of the next stage.

```
ENTITY fig6_22 IS
PORT(
   cin   :IN BIT;
   a     :IN BIT_VECTOR(3 DOWNTO 0);
   b     :IN BIT_VECTOR(3 DOWNTO 0);
   s     :OUT BIT_VECTOR(3 DOWNTO 0);
   cout  :OUT BIT);
END fig6_22;

ARCHITECTURE a OF fig6_22 IS
SIGNAL c :BIT_VECTOR (4 DOWNTO 0);  -- carries require 5 bit array

BEGIN
   c(0) <= cin;         -- Read the carry in to bit array
   s <= a XOR b  XOR c(3 DOWNTO 0); -- Generate the sum bits
   c(4 DOWNTO 1) <=     (a AND b)
                  OR    (a AND c(3 DOWNTO 0))
                  OR    (b AND c(3 DOWNTO 0));
   cout <= c(4);        -- output the carry of the MSB.
END a;
```

FIGURE 6-22 VHDL adder.

REVIEW QUESTIONS

1. If $[A] = 1001$ and $[B] = 0011$, what is the value of (a) $[A] \cdot [B]$? (b) $[A] + [B]$?(Note that $\cdot$ means AND; $+$ means OR.)
2. If $A[7..0] = 1010\ 1100$, what is the value of (a) $A[7..4]$? (b) $A[5..2]$?
3. In AHDL, the following object is declared: toggles[7..0] :INPUT. Give an expression for the least significant four bits using AHDL syntax.
4. In VHDL, the following object is declared: toggles :IN BIT_VECTOR (7 DOWNTO 0). Give an expression for the least significant four bits using VHDL syntax.
5. What would be the result of ORing the two registers of Example 6-14?
6. Write an HDL statement that would OR the two objects *d* and *g* together. Use your favorite HDL.
7. Write an HDL statement that would XOR the two most significant bits of *d* with the two least significant bits of *g* and put the result in the middle two bits of *x*.

6-21 EXPANDING THE BIT CAPACITY OF A CIRCUIT

One way we have learned to expand the capacity of a circuit is to cascade stages, like we did with the 74382 ALU chip in the previous section. This can be done using the Altera graphic design file approach (like Figure 6-18) or the structural text-based HDL approach. With either of these methods, we need to specify all the inputs, outputs, and interconnections between blocks. In the case of this adder circuit, it would be much easier to start with the HDL file for a four-bit adder and simply increase the size of each operand variable in the equation. For example, if we wanted an eight-bit adder, we simply need to expand *a*, *b*, and *s* to eight bits. The code would remain almost identical to the four-bit adder shown above. This is just a glimpse of some of the efficiency improvements that HDL offers. The way this code is written, however, the indices of each signal, and each *bit array* specification in the equation, would also have to be redefined. In other words, the designer would need to examine the code carefully and change all the 3s to 7s, all the 4s to 8s, and so on.

An important principle in software engineering is symbolic representation of the **constants** that are used throughout the code. Constants are simply fixed numbers represented by a name (symbol). If we can define a symbol (i.e., make up a name) at the top of the source code that is assigned the value for the total number of bits and then use this symbol (name) throughout the code, it is much easier to modify the circuit. Only one line of code needs to be changed to expand the capacity of the circuit. The examples that follow add this feature to the code and also upgrade the code to implement the adder/subtractor circuit like the one in Figure 6-14. It should be noted here that expanding the capacity of an adder circuit such as this one will also reduce the speed of the circuit because of carry propagation (described in Section 6-13). In order to keep these examples simple, we have not added any logic to generate a look-ahead carry.

AHDL ADDER/SUBTRACTOR

In AHDL, using constants is very simple, as shown on lines 1 and 2 of Figure 6-23. The keyword CONSTANT is followed by the symbolic name and the value it is to be assigned. Notice that we can allow the compiler to do some

```
 1   CONSTANT number_of_bits = 8;                 -- set total number of bits
 2   CONSTANT n = number_of_bits - 1;             -- n is highest bit index
 3
 4   SUBDESIGN fig6_23
 5   (
 6      add          :INPUT;      -- add control
 7      sub          :INPUT;      -- subtract control and LSB Carry in
 8      a[n..0]      :INPUT;      -- Augend bits
 9      bin[n..0]    :INPUT;      -- Addend bits
10      s[n..0]      :OUTPUT;     -- Sum bits
11      caryout      :OUTPUT;     -- MSB carry OUT
12   )
13   VARIABLE
14      c[n+1..0]   :NODE;       -- intermediate carry vector
15      b[n..0]      :NODE;       -- intermediate operand vector
16   BEGIN
17      b[] = bin[] & add # NOT bin[] & sub;
18      c[0] = sub;                      --Read the carry in to group variable
19      s[] = a[] $ b[] $ c[n..0];       --Generate the sums
20      c[n+1..1] = (a[] & b[]) # (a[] & c[n..0]) # (b[] & c[n..0]);
21      caryout = c[n+1];                -- output the carry of the MSB.
22   END;
```

FIGURE 6-23 An *n*-bit adder/subtractor description in AHDL.

simple math calculations to establish a value for one constant based on another. We can also use this feature as we refer to the constant in the code, as shown on lines 14, 20, and 21. For example, we can refer to *c[7]* as *c[n]* and *c[8]* as *c[n+1]*. The size of this adder/subtractor can be expanded by simply changing the value of *number_of_bits* to the desired number of bits and then recompiling.

As mentioned, this code has been upgraded from the previous example to make it an adder/subtractor like the one in Figure 6-14. The *add* and *sub* inputs have been included on lines 6 and 7, and a new intermediate variable named *b[ ]* has been included on line 15. The first concurrent statement on line 17 describes all the SOP logic that drives the *b* inputs to the adder in Figure 6-14. First, it describes a logical AND operation between every bit of *bin[ ]* and the logic level on *add.* This result is ORed (bit for bit) with the result of ANDing the complement of every bit of *bin[ ]* with *sub.* In other words, it creates the following Boolean function for each bit: $b = bin \cdot add + \overline{bin} \cdot sub$. The signal *b[ ]* is then used in the adder equations instead of *bin[ ],* as was used in the adder examples. Notice on line 18 that *sub* is also used to connect the carry array LSB (carry into bit 0) with the value on *sub*, which needs to be 0 when adding and 1 when subtracting.

VHDL ADDER/SUBTRACTOR

In VHDL, using constants is a little bit more involved. Constants must be included in a **PACKAGE**, as shown in Figure 6-24, lines 1–4. Packages are also used to contain component definitions and other information that must be available to all entities in the design file. Notice on line 6 that the keyword USE tells the compiler to use the definitions in this package throughout this design file. Inside the package, the keyword CONSTANT is followed by the symbolic name, its type, and the value it is to be assigned using the := operator. Notice on line 3 that we can allow the compiler to do some simple math calculations to establish a value of one constant based on another. We can also use this feature as we refer to the constant in the code, as shown on lines 34 and 37. For example, we can refer to *c(7)* as *c(n)* and *c(8)* as *c(n+1)*. The size of this adder/subtractor can be expanded by simply changing the value of *number_of_bits* to the desired number of bits and then recompiling.

As mentioned, this code has been upgraded from the previous example to make it an adder/subtractor like the one in Figure 6-14. The *add* and *sub* inputs have been included on lines 10 and 11, and a new signal named *b* has been included on line 20, *bnot* has been included on line 21, and *mode* has been included on line 22. The first concurrent statement on line 24 serves to create the 1's complement of *bin.* The SOP circuits in Figure 6-14 that drive the *b* inputs to the adder select the *bin* inputs if *add* = 1 or the 1's complement (*bnot*) if *sub* = 1. This is an excellent application of the VHDL selected signal assignment, as shown on lines 27–30. When *add* is 1, *bin* is channeled to *b.* When *sub* is 1, *bnot* is channeled to *b.* The signal *b* is then used in the adder equations instead of *bin,* as was used in the previous adder examples. Notice on line 32 that *sub* is also used to connect the carry array LSB (carry into bit 0) with the value on *sub,* which needs to be 0 when adding and 1 when subtracting.

```
PACKAGE const IS
  CONSTANT number_of_bits :INTEGER:=8;  -- set total number of bits
  CONSTANT n :INTEGER:= number_of_bits - 1;  -- MSB index number
END const;

USE work.const.all;

ENTITY fig6_24 IS
PORT(
   add      :IN BIT; -- add control
   sub      :IN BIT; -- subtract control and LSB carry in
   a        :IN BIT_VECTOR(n DOWNTO 0);
   bin      :IN BIT_VECTOR(n DOWNTO 0);
   s        :OUT BIT_VECTOR(n DOWNTO 0);
   carryout :OUT BIT);
END fig6_24;

ARCHITECTURE a OF fig6_24 IS
SIGNAL c :BIT_VECTOR (n+1 DOWNTO 0);  -- define intermediate carries
SIGNAL b :BIT_VECTOR (n DOWNTO 0);     -- define intermediate operand
SIGNAL bnot :BIT_VECTOR (n DOWNTO 0);
SIGNAL mode :BIT_VECTOR (1 DOWNTO 0);
BEGIN
   bnot <= NOT bin;
   mode <= add & sub;

   WITH mode SELECT
      b <= bin       WHEN “10”,    -- add
           bnot      WHEN “01”,    -- sub
           “0000”    WHEN OTHERS;

   c(0) <= sub;                    -- read the carry_in to bit array
   s <= a XOR b XOR c(n DOWNTO 0); -- generate the sum bits
   c(n+1 DOWNTO 1) <= (a AND b) OR
                       (a AND c(n DOWNTO 0)) OR
                       (b AND c(n DOWNTO 0)); --generate carries
   carryout <= c(n+1);         -- output the carry of the MSB.

END a;
```

FIGURE 6-24 An *n*-bit adder/subtractor description in VHDL.

VHDL GENERATE Statement

Another way to make circuits that handle more bits is the VHDL **GENERATE** statement. It is a very concise way of telling the compiler to replicate several components that are cascaded together. As we have shown, there are many other ways to accomplish the same thing and if the abstract nature of this method seems difficult, use another method. The GENERATE statement is offered here for the sake of completeness. The adder circuits we have been discussing are cascaded chains of single-bit full adder modules. The VHDL code for a single-bit full adder module is shown in Figure 6-25. Multiple instances of this module need to be connected to each other to form

FIGURE 6-25 Single-bit full adder in VHDL.

```
ENTITY add1 IS
PORT(
      cin   :IN BIT;
      a     :IN BIT;
      b     :IN BIT;
      s     :OUT BIT;
      cout  :OUT BIT);
END add1;

ARCHITECTURE a OF add1 IS
BEGIN

     s      <= a XOR b XOR cin;
     cout   <= (a AND b) OR (a AND cin) OR (b AND cin);
END a;
```

an *n*-bit adder circuit. Of course, you can do it using the same component techniques that we have discussed previously, but it would result in very lengthy code.

To make the code more concise and easier to modify, a strategy is needed for the way we label the inputs and outputs for each module. As we mentioned previously, the bit-0 adder has inputs that have an index of 0 (e.g., *a0*, *b0*, *c0*, *s0*). The carry out of bit 0 is labeled *c1*, and it becomes the carry input for the bit 1 adder module. Each time we instantiate another component for the next bit of the multibit adder, the index number of all connections goes up by 1 (*a1*, *b1*, *c1*, *s1*). The GENERATE statement allows us to repeat an instantiation of a component *n* times, increasing the index number by 1 for each instantiation up to *n*. In line 27 of Figure 6-26, the GENERATE keyword is used in an **iterative loop (FOR loop)**, which means that a set of descriptive actions (PORT MAP) will be repeated a certain number of times. The variable *i* represents an index number that starts at 0 (for the first iteration) and ends at *n* (the last iteration). The advantages of this method are code compactness and the ease with which the number of bits can be expanded. The code in Figure 6-26 shows how to use a single-bit adder (Figure 6-25) as a component to generate an eight-bit adder circuit. Remember that the file for the single-bit adder (add1.vhd in Figure 6-25) must be saved in the same folder as the design file that uses it to generate multiple instances of the adder (fig6_26.vhd).

The single-bit adder component is defined on lines 17–23 of Figure 6-26. For the first iteration of lines 29 and 30, the value of *i* is 0, creating an adder stage for bit 0. The second iteration, *i*, has been increased to 1 to form adder stage 1. This continues until *i* is equal to *n*, generating each stage of the (n + 1)-bit adder. Notice that VHDL allows us the option of placing labels at the beginning of a line of code to help describe its purpose. For example, on line 27, the label *repeat* is used and on line 29, the label *casc* is used. Labels are optional but they must always end with a colon.

Libraries of Parameterized Modules

Using HDL techniques clearly makes it easy to alter the bit capacity of a generic circuit. In this chapter, we can see that it is easy to change from a

```
PACKAGE const IS
   CONSTANT number_of_bits :INTEGER:=8;   -- Specify number of bits
   CONSTANT n :INTEGER:=number_of_bits - 1;     --n is MSB bit number
END const;
USE work.const.all;
ENTITY fig6_26 IS
PORT(
   caryin   :IN bit;
   ain      :IN BIT_VECTOR (n DOWNTO 0);
   bin      :IN BIT_VECTOR (n DOWNTO 0);
   sout     :OUT BIT_VECTOR (n DOWNTO 0);
  carryout  :OUT bit);
END fig6_26;

ARCHITECTURE a OF fig6_26 IS

COMPONENT add1          -- declare single bit full adder
   PORT  (
            cin   :IN BIT;
            a,b   :IN BIT;
            s     :OUT BIT;
            cout  :OUT BIT);
END COMPONENT;
SIGNAL c :BIT_VECTOR (n+1 DOWNTO 0);   -- declare bit array for carries
BEGIN
   c(0)  <= caryin;                    -- put LSB in array (carry in)
   repeat:FOR i IN 0 TO n GENERATE     -- instantiate n+1 adders
         -- cascade them
   casc:add1 PORT MAP (cin=> c(i), a=> ain(i), b=> bin(i),
                       s=> sout(i), cout=> c(i+1));
   END GENERATE;
   carryout    <= c(n+1);            -- out the carry from nth bit stage
END a;
```

FIGURE 6-26 Use of the VHDL GENERATE statement.

four-bit adder to an eight-, 12-, or 16-bit adder. When Altera was creating its library of useful functions, they also took advantage of these techniques and created what they refer to as **megafunctions**, which include a **library of parameterized modules (LPMs)**. These functions do not attempt to imitate a particular standard IC like the old-style macrofunctions; instead, they offer a generic solution for the various types of logic circuits that are useful in digital systems. Examples of these generic circuits that we have covered so far are logic gates (AND, OR, XOR), latches, counters, shift registers, and adders. The term *parameterized* means that when you instantiate a function from the library, you also specify some parameters that define certain attributes (bit capacity, for example) for the circuit you are describing. The various LPMs that are available can be found through the HELP menu under megafunctions/LPM. This documentation describes the parameters that the user can specify as well as the port features of the device.

REVIEW QUESTIONS

1. What keyword is used to assign a symbolic name to a fixed number?
2. In AHDL, where are constants defined? Where are they defined in VHDL?
3. Why are constants useful?
4. If the constant max_val has a value of 127, how will a compiler interpret the expression max_val −5?
5. What is the GENERATE statement used for in VHDL?

SUMMARY

1. To represent signed numbers in binary, a sign bit is attached as the MSB. A + sign is a 0, and a − sign is a 1.
2. The 2's complement of a binary number is obtained by complementing each bit and then adding 1 to the result.
3. In the 2's-complement method of representing signed binary numbers, positive numbers are represented by a sign bit of 0 followed by the magnitude in its true binary form. Negative numbers are represented by a sign bit of 1 followed by the magnitude in 2's-complement form.
4. A signed binary number is negated (changed to a number of equal value but opposite sign) by taking the 2's complement of the number, including the sign bit.
5. Subtraction can be performed on signed binary numbers by negating (2's complementing) the subtrahend and adding it to the minuend.
6. In BCD addition, a special correction step is needed whenever the sum of a digit position exceeds 9 (1001).
7. When signed binary numbers are represented in hexadecimal, the MSD of the hex number will be 8 or greater when the number is negative; it will be 7 or less when the number is positive.
8. The arithmetic/logic unit (ALU) of a computer contains the circuitry needed to perform arithmetic and logic operations on binary numbers stored in memory.
9. The accumulator is a register in the ALU. It holds one of the numbers being operated upon, and it also is where the result of the operation is stored in the ALU.
10. A full adder performs the addition on two bits plus a carry input. A parallel binary adder is made up of cascaded full adders.
11. The problem of excessive delays caused by carry propagation can be reduced by a look-ahead carry logic circuit.
12. IC adders such as the 74LS83/HC83 and the 74LS283/HC283 can be used to construct high-speed parallel adders and subtractors.
13. A BCD adder circuit requires special correction circuitry.
14. Integrated-circuit ALUs are available that can be commanded to perform a wide range of arithmetic and logic operations on two input numbers.
15. Prefabricated functions are available in the Altera libraries.
16. These library parts and the HDL circuits you create can be interconnected using either graphic or structural HDL techniques.

17. Logical operations can be performed on all the bits in a set using Boolean equations.
18. Practicing good software engineering techniques, specifically the use of symbols to represent constants, allows for easy code modification and expansion of the bit capacity of circuits such as full adders.
19. Libraries of parameterized modules (LPMs) offer a flexible, easily modified or expanded solution for many types of digital circuits.

IMPORTANT TERMS

carry
sign bit
sign-magnitude system
2's-complement system
negation
augend
addend
subtrahend
minuend
overflow
arithmetic/logic unit (ALU)
accumulator register
full adder (FA)
parallel adder
half adder (HA)
carry propagation (carry ripple)
look-ahead carry
adder/subtractor
macrofunction
function prototype
sets
constants
PACKAGE
GENERATE
iterative loop
FOR loop
library of parameterized functions (LPMs)
megafunctions

PROBLEMS

SECTION 6-1

B 6-1. Add the following in binary. Check your results by doing the addition in decimal.

(a)* 1010 + 1011
(b)* 1111 + 0011
(c)* 1011.1101 + 11.1
(d) 0.1011 + 0.1111
(e) 10011011 + 10011101
(f) 1010.01 + 10.111

SECTION 6-2

B 6-2. Represent each of the following signed decimal numbers in the 2's-complement system. Use a total of eight bits, including the sign bit.

(a)* +32
(b)* −14
(c)* +63
(d)* −104
(e)* +127
(f)* −127
(g)* +89
(h)* −55
(i) −1
(j) −128
(k) +169
(l) 0
(m) +84
(n) +3
(o) −3
(p) −190

B 6-3. Each of the following numbers represents a signed decimal number in the 2's-complement system. Determine the decimal value in each case. (*Hint:* Use negation to convert negative numbers to positive.)

(a)* 01101
(b)* 11101
(c)* 01111011
(d)* 10011001
(e)* 01111111
(f) 10000000
(g) 11111111
(h) 10000001
(i) 01100011
(j) 11011001

*Answers to problems marked with an asterisk can be found in the back of the text.

6-4. (a) What range of signed decimal values can be represented using 12 bits, including the sign bit?

(b) How many bits would be required to represent decimal numbers from −32,768 to +32,767?

6-5.* List, in order, all of the signed numbers that can be represented in five bits using the 2's-complement system.

6-6. Represent each of the following decimal values as an eight-bit signed binary value. Then negate each one.

(a)* +73 (b)* −12 (c) +15 (d) −1 (e) −128 (f) +127

6-7. (a)* What is the range of unsigned decimal values that can be represented in 10 bits? What is the range of signed decimal values using the same number of bits?

(b) Repeat both problems using eight bits.

SECTIONS 6-3 AND 6-4

6-8. The reason why the sign-magnitude method for representing signed numbers is not used in most computers can readily be illustrated by performing the following.

(a) Represent +12 in eight bits using the sign-magnitude form.

(b) Represent −12 in eight bits using the sign-magnitude form.

(c) Add the two binary numbers and note that the sum does not look anything like zero.

6-9. Perform the following operations in the 2's-complement system. Use eight bits (including the sign bit) for each number. Check your results by converting the binary result back to decimal.

(a)* Add +9 to +6.
(b)* Add +14 to −17.
(c)* Add +19 to −24.
(d)* Add −48 to −80.
(e)* Subtract +16 from +17.
(f) Subtract +21 from −13.
(g) Subtract +47 from +47.
(h) Subtract −36 from −15.
(i) Add +17 to −17.
(j) Subtract −17 from −17.

6-10. Repeat Problem 6-9 for the following cases, and show that overflow occurs in each case.

(a) Add +37 to +95.
(b) Subtract +37 from −95.
(c) Add −37 to −95.
(d) Subtract −37 from +95.

SECTIONS 6-5 AND 6-6

B 6-11. Multiply the following pairs of binary numbers, and check your results by doing the multiplication in decimal.

(a)* 111 × 101
(b)* 1011 × 1011
(c) 101.101 × 110.010
(d) .1101 × .1011

B 6-12. Perform the following divisions. Check your results by doing the division in decimal.

(a)* 1100 ÷ 100
(b)* 111111 ÷ 1001
(c) 10111 ÷ 100
(d) 10110.1101 ÷ 1.1

SECTIONS 6-7 AND 6-8

B 6-13. Add the following decimal numbers after converting each to its BCD code.

(a)* 74 + 23	(d) 385 + 118
(b)* 58 + 37	(e) 998 + 003
(c)* 147 + 380	(f) 623 + 599

B 6-14. Find the sum of each of the following pairs of hex numbers.

(a)* 3E91 + 2F93	(d) 2FFE + 0002
(b)* 91B + 6F2	(e) FFF + 0FF
(c)* ABC + DEF	(f) D191 + AAAB

B 6-15. Perform the following subtractions on the pairs of hex numbers.

(a)* 3E91 − 2F93	(d) 0200 − 0003
(b)* 91B − 6F2	(e) F000 − EFFF
(c)* 0300 − 005A	(f) 2F00 − 4000

6-16. The owner's manual for a small microcomputer states that the computer has usable memory locations at the following hex addresses: 0200 through 03FF, and 4000 through 7FD0. What is the total number of available memory locations?

6-17. (a)* A certain eight-bit memory location holds the hex data 77. If this represents an *unsigned* number, what is its decimal value?

(b)* If this represents a *signed* number, what is its decimal value?

(c) Repeat (a) and (b) if the data value is E5.

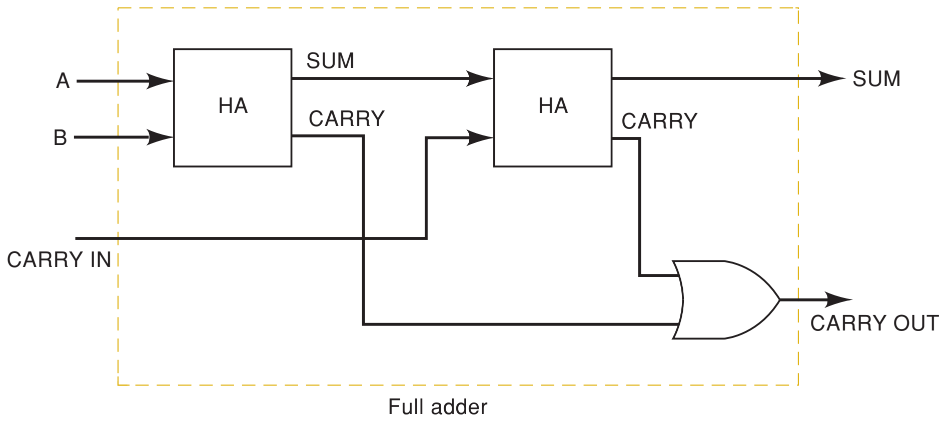
SECTION 6-11

6-18. Convert the FA circuit of Figure 6-8 to all NAND gates.

6-19.* Write the function table for a half adder (inputs *A* and *B*; outputs SUM and CARRY). From the function table, design a logic circuit that will act as a half adder.

6-20. A full adder can be implemented in many different ways. Figure 6-27 shows how one may be constructed from two half adders. Construct a function table for this arrangement, and verify that it operates as a FA.

FIGURE 6-27 Problem 6-20.



SECTION 6-12

6-21.* Refer to Figure 6-10. Determine the contents of the A register after the following sequence of operations: $[A] = 0000$, $[0100] \rightarrow [B]$, $[S] \rightarrow [A]$, $[1011] \rightarrow [B]$, $[S] \rightarrow [A]$.

6-22. Refer to Figure 6-10. Assume that each FF has $t_{PLH} = t_{PHL} = 30$ ns and a setup time of 10 ns, and that each FA has a propagation delay of 40 ns. What is the minimum time allowed between the PGT of the LOAD pulse and the PGT of the TRANSFER pulse for proper operation?

D 6-23. In the adder and subtractor circuits discussed in this chapter, we gave no consideration to the possibility of *overflow*. Overflow occurs when the two numbers being added or subtracted produce a result that contains more bits than the capacity of the accumulator. For example, using four-bit registers, including a sign bit, numbers ranging from $+7$ to -8 (in 2's complement) can be stored. Therefore, if the result of an addition or subtraction exceeds $+7$ or -8, we would say that an overflow has occurred. When an overflow occurs, the results are useless because they cannot be stored correctly in the accumulator register. To illustrate, add $+5$ (0101) and $+4$ (0100), which results in 1001. This 1001 would be interpreted incorrectly as a negative number because there is a 1 in the sign-bit position.

In computers and calculators, there are usually circuits that are used to detect an overflow condition. There are several ways to do this. One method that can be used for the adder that operates in the 2's-complement system works as follows:

1. Examine the sign bits of the two numbers being added.
2. Examine the sign bit of the result.
3. Overflow occurs whenever the numbers being added are *both positive* and the sign bit of the result is 1 *or* when the numbers are *both negative* and the sign bit of the result is 0.

This method can be verified by trying several examples. Readers should try the following cases for their own clarification: (1) $5 + 4$; (2) $-4 + (-6)$; (3) $3 + 2$. Cases 1 and 2 will produce an overflow, and case 3 will not. Thus, by examining the sign bits, one can design a logic circuit that will produce a 1 output whenever the overflow condition occurs. Design this overflow circuit for the adder of Figure 6-10.

C, D 6-24. Add the necessary logic circuitry to Figure 6-10 to accommodate the transfer of data from memory into the A register. The data values from memory are to enter the A register through its D inputs on the PGT of the *first* TRANSFER pulse; the data from the sum outputs of the FAs will be loaded into A on the PGT of the *second* TRANSFER. In other words, a LOAD pulse followed by two TRANSFER pulses is required to perform the complete sequence of loading the B register from memory, loading the A register from memory, and then transferring their sum into the A register. (*Hint:* Use a flip-flop X to control which source of data gets loaded into the D inputs of the accumulator.)

SECTION 6-13

C, D 6-25.* Design a look-ahead carry circuit for the adder of Figure 6-10 that generates the carry C_3 to be fed to the FA of the MSB position based on the values of A_0, B_0, C_0, A_1, B_1, A_2, and B_2. In other words, derive an expression for C_3 in terms of A_0, B_0, C_0, A_1, B_1, A_2, and B_2. (*Hint:* Begin by writing the expression for C_1 in terms of A_0, B_0, and C_0. Then write the expression for C_2 in terms of A_1, B_1, and C_1. Substitute the expression for C_1 into the expression for C_2. Then write the expression for C_3 in terms of A_2, B_2, and C_2. Substitute the expression for C_2 into the expression for C_3. Simplify the final expression for C_3 and put it in sum-of-products form. Implement the circuit.)

SECTION 6-14

6-26. Show the logic levels at each input and output of Figure 6-11(a) when 354_8 is added to 103_8.

SECTION 6-15

6-27. For the circuit of Figure 6-14, determine the sum outputs for the following cases.

(a)* A register = 0101 (+5), B register = 1110 (−2); SUB = 1, ADD = 0

(b) A register = 1100 (−4), B register = 1110 (−2); SUB = 0, ADD = 1

(c) Repeat (b) with ADD = SUB = 0.

6-28. For the circuit of Figure 6-14 determine the sum outputs for the following cases.

(a) A register = 1101 (−3), B register = 0011 (+3), SUB = 1, ADD = 0.

(b) A register = 1100 (−4), B register = 0010 (+2), SUB = 0, ADD = 1.

(c) A register = 1011 (−5), B register = 0100 (+4), SUB = 1, ADD = 0.

6-29. For each of the calculations of Problem 6-27, determine if overflow has occurred.

6-30. For each of the calculations of Problem 6-28, determine if overflow has occurred.

D 6-31. Show how the gates of Figure 6-14 can be implemented using three 74HC00 chips.

D 6-32.* Modify the circuit of Figure 6-14 so that a single control input, X, is used in place of ADD and SUB. The circuit is to function as an adder when $X = 0$, and as a subtractor when $X = 1$. Then simplify each set of gates. (*Hint:* Note that now each set of gates is functioning as a controlled inverter.)

SECTION 6-16

B 6-33. Determine the F, C_{N+4}, and OVR outputs for each of the following sets of inputs applied to a 74LS382.

(a)* $[S] = 011$, $[A] = 0110$, $[B] = 0011$, $C_N = 0$

(b) $[S] = 001$, $[A] = 0110$, $[B] = 0011$, $C_N = 1$

(c) $[S] = 010$, $[A] = 0110$, $[B] = 0011$, $C_N = 1$

D 6-34. Show how the 74HC382 can be used to produce $[F] = [\overline{A}]$. (*Hint:* Recall that special property of an XOR gate.)

6-35. Determine the Σ outputs in Figure 6-16 for the following sets of inputs.

(a)* $[S] = 110$, $[A] = 10101100$, $[B] = 00001111$

(b) $[S] = 100$, $[A] = 11101110$, $[B] = 00110010$

C, D 6-36. Add the necessary logic to Figure 6-16 to produce a single HIGH output whenever the binary number at A is exactly the same as the binary number at B. Apply the appropriate select input code (three codes can be used).

SECTION 6-17

T 6-37. Consider the circuit of Figure 6-10. Assume that the A_2 output is stuck LOW. Follow the sequence of operations for adding two numbers, and determine the results that will appear in the A register after the second TRANSFER pulse for each of the following cases. Note that the numbers are given in decimal, and the first number is the one loaded into B by the first LOAD pulse.

(a)* 2 + 3

(b)* 3 + 7

T (c) 7 + 3

(d) 8 + 3

(e) 9 + 3

T 6-38. A technician breadboards the adder/subtractor of Figure 6-14. During testing, she finds that whenever an addition is performed, the result is 1 more than expected, and when a subtraction is performed, the result is 1 less than expected. What is the likely error that the technician made in connecting this circuit?

6-39.* Describe the symptoms that would occur at the following points in the circuit of Figure 6-14 if the ADD and SUB lines were shorted together.

(a) B[3..0] inputs of the 74LS283 IC

(b) C_0 input of the 74LS283 IC

(c) SUM ($\sum$) [3..0] outputs

(d) C_4

SECTION 6-19

Problems 6-40 through 6-45 deal with the same two arrays, a and b, which we will assume have been defined in an HDL source file and have the following values: [a] = [10010111], [b] = [00101100]. Output array [z] is also an eight-bit array. Answer Problems 6-40 through 6-45 based on this information. (Assume undefined bits in z are 0.)

B, H 6-40. Declare these data objects using your favorite HDL syntax.

B, H 6-41. Give the value of z for each expression (identical AHDL and VHDL expressions are given):

(a)* z[] = a[] & b[];	z <= a AND b;
(b)* z[] = a[] # b[];	z <= a OR b;
(c) z[] = a[] $!b[];	z <= a XOR NOT b;

(d) z[7..4] = a[3..0] & b[3..0]; z(7 DOWNTO 4) <= a(3 DOWNTO 0) AND b(3 DOWNTO 0);

(e) z[7..1] = a[6..0]; z[0] = GND; z(7 DOWNTO 1) <= a(6 DOWNTO 0); z(0) <= ‘0’;

6-42. What is the value of each of the following:

(a) a[3..0]	a(3 DOWNTO 0)
(b) b[0]	b(0)
(c) a[7]	b(7)

B, H 6-43. What is the value of each of the following?

(a)* a[5]	a(5)
(b)* b[2]	b(2)
(c)* b[7..1]	b(7 DOWNTO 1)

H 6-44.* Write one or more statements in HDL that will shift all the bits in [a] one position to the right. The LSB should move to the MSB position. The rotated data should end up in z[ ].

6-45. Write one or more HDL statements that will take the upper nibble of b and place it in the lower nibble of z. The upper nibble of z should be zero.

D, H 6-46. Refer to Problem 6-23. Modify the code of Figure 6-21 or Figure 6-22 to add an overflow output.

D, H, N 6-47.* Another way to detect 2's-complement overflow is to XOR the carry into the MSB with the carry out of the MSB of an adder/subtractor. Use the same numbers given in Problem 6-23 to verify this. Modify Figure 6-21 or Figure 6-22 to detect overflow using this method.

D, H 6-48.* Modify Figure 6-21 or Figure 6-22 to implement Figure 6-10.

SECTION 6-20

B, H 6-49. Modify Figure 6-21 or Figure 6-22 to make it a 12-bit adder without using constants.

B, H 6-50. Modify Figure 6-21 or Figure 6-22 to make it a versatile *n*-bit adder module with a constant defining the number of bits.

D, C, H 6-51. Write an HDL file to create the equivalent of a 74382 ALU without using a built-in macrofunction.

DRILL QUESTION

6-52. Define each of the following terms.

(a) Full adder	(f) Accumulator
(b) 2's complement	(g) Parallel adder
(c) Arithmetic/logic unit	(h) Look-ahead carry
(d) Sign bit	(i) Negation
(e) Overflow	(j) *B* register

MICROCOMPUTER APPLICATIONS

C, D 6-53.* In a typical microprocessor ALU, the results of every arithmetic operation are usually (but not always) transferred to the accumulator register, as in Figures 6-10, 6-14, and 6-15. In most microprocessor ALUs, the result of each arithmetic operation is also used to control the states of several special flip-flops called *flags.* These flags are used by the microprocessor when it is making decisions during the execution of certain types of instructions. The three most common flags are:

S (sign flag). This FF is always in the same state as the sign of the last result from the ALU.

Z (zero flag). This flag is set to 1 whenever the result from an ALU operation is exactly 0. Otherwise, it is cleared to 0.

C (carry flag). This FF is always in the same state as the carry from the MSB of the ALU.

Using the adder/subtractor of Figure 6-14 as the ALU, design the logic circuit that will implement these flags. The sum outputs and C_4 output are to be used to control what state each flag will go to upon the occurrence of the TRANSFER pulse. For example, if the sum is exactly 0 (i.e., 0000), the Z flag should be set by the PGT of TRANSFER; otherwise, it should be cleared.

6-54.*In working with microcomputers, it is often necessary to move binary numbers from an eight-bit register to a 16-bit register. Consider the numbers 01001001 and 10101110, which represent +73 and −82, respectively, in the 2's-complement system. Determine the 16-bit representations for these decimal numbers.

6-55. Compare the eight- and 16-bit representations for +73 from Problem 6-53. Then compare the two representations for −82. There is a general rule that can be used to convert easily from eight-bit to 16-bit representations. Can you see what it is? It has something to do with the sign bit of the eight-bit number.

ANSWERS TO SECTION REVIEW QUESTIONS

SECTION 6-1

1. (a) 11101 (b) 101.111 (c) 10010000

SECTION 6-2

1. (a) 00001101 (b) 11111001 (c) 10000000 2. (a) −29 (b) −64 (c) +126 3. −2048 to +2047 4. Seven 5. −32768 6. (a) 10000 (b) 10000000 (c) 1000 7. Refer to text.

SECTION 6-3

1. True 2. (a) $100010_2 = -30_{10}$ (b) $000000_2 = 0_{10}$

SECTION 6-4

1. (a) $01111_2 = +15_{10}$ (b) $11111_2 = -1_{10}$ 2. By comparing the sign bit of the sum with the sign bits of the numbers being added

SECTION 6-5

1. 1100010

SECTION 6-7

1. The sum of at least one decimal digit position is greater than 1001 (9).
2. The correction factor is added to both the units and the tens digit positions.

SECTION 6-8

1. 923 2. 3DB 3. 2F, 77EC, 6D

SECTION 6-10

1. Three; two 2. (a) $S_2 = 0, C_3 = 1$ (b) $C_5 = 0$

SECTION 6-12

1. One; four; four 2. 0100

SECTION 6-14

1. Five chips 2. 240 ns 3. 1

SECTION 6-15

1. To add the 1 needed to complete the 2's-complement representation of the number in the *B* register 2. 0010 3. 1101 4. False; the 1's complement appears there.

SECTION 6-16

1. $F = 1011$; $OVR = 0$; $C_{N+4} = 0$ 2. $F = 0111$; $OVR = 1$; $C_{N+4} = 1$ 3. $F = 1000$
4. $\Sigma = 01101011$; $C_{N+4} = OVR = 0$ 5. $\Sigma = 11111111$ 6. Eight

SECTION 6-18

1. See the MAX+PLUS HELP menu under old-style macrofunctions/adders.
2. An HDL description of a standard IC that can be used from the library.

SECTION 6-19

1. (a) 0001 (b) 1011 2. (a) 1010 (b) 1011 3. toggles[3..0]
4. toggles(3 DOWNTO 0) 5. [X] = [1,1,1,1] 6. AHDL: xx[] = d[] # g[];
VHDL: x <= d OR g; 7. AHDL: xx[2..1] = d[3..2] $ g[1..0]; VHDL:
x(2 DOWNTO 1) <= d(3 DOWNTO 2) XOR g(1 DOWNTO 0);

SECTION 6-21

1. CONSTANT. 2. In AHDL, near the top of the source file. In VHDL, in a PACKAGE near the top of the source file. 3. They allow for global changes of the value of a symbol used throughout the code. 4. max_val −5 represents the number 122. 5. GENERATE is used with an iterative FOR statement to instantiate duplicate code modules that can be connected together or cascaded.