CHAPTER 2

Logic Circuits

Many important functions are naturally computed with **straight-line programs**, programs without loops or branches. Such computations are conveniently described with **circuits**, directed acyclic graphs of straight-line programs. Circuit vertices are associated with program steps, whereas edges identify dependencies between steps. Circuits are characterized by their **size**, the number of vertices, and their **depth**, the length (in edges) of their longest path. Circuits in which the operations are Boolean are called **logic circuits**, those using algebraic operations are called **algebraic circuits**, and those using comparison operators are called **comparator circuits**. In this chapter we examine logic circuits. Algebraic and comparator circuits are examined in Chapter 6.

Logic circuits are the basic building blocks of real-world computers. As shown in Chapter 3, all machines with bounded memory can be constructed of logic circuits and binary memory units. Furthermore, machines whose computations terminate can be completely simulated by circuits.

In this chapter circuits are designed for a large number of important functions. We begin with a discussion of circuits, straight-line programs, and the functions computed by them. Normal forms, a structured type of circuit, are examined next. They are a starting point for the design of circuits that compute functions. We then develop simple circuits that combine and select data. They include logical circuits, encoders, decoders, multiplexers, and demultiplexers. This is followed by an introduction to prefix circuits that efficiently perform running sums. Circuits are then designed for the arithmetic operations of addition (in which prefix computations are used), subtraction, multiplication, and division. We also construct efficient circuits for symmetric functions. We close with proofs that every Boolean function can be realized with size and depth exponential and linear, respectively, in its number of inputs, and that most Boolean functions require such circuits.

The concept of a reduction from one problem to a previously solved one is introduced in this chapter and applied to many simple functions. This important idea is used later to show that two problems, such as different **NP**-complete problems, have the same computational complexity. (See Chapters 3 and 8.)

2.1 Designing Circuits

The logic circuit, as defined in Section 1.4.1, is a directed acyclic graph (DAG) whose vertices are labeled with the names of Boolean functions (logic gates) or variables (inputs). Each logic circuit computes a binary function $f : \mathcal{B}^n \mapsto \mathcal{B}^m$ that is a mapping from the values of its n input variables to the values of its m outputs.

Computer architects often need to design circuits for functions, a task that we explore in this chapter. The goal of the architect is to design efficient circuits, circuits whose size (the number of gates) and/or depth (the length of the longest path from an input to an output vertex) is small. The computer scientist is interested in circuit size and depth because these measures provide lower bounds on the resources needed to complete a task. (See Section 1.5.1 and Chapter 3.) For example, circuit size provides a lower bound on the product of the space and time needed for a problem on both the random-access and Turing machines (see Sections 3.6 and 3.9.2) and circuit depth is a measure of the parallel time needed to compute a function (see Section 8.14.1).

The logic circuit also provides a framework for the classification of problems by their computational complexity. For example, in Section 3.9.4 we use circuits to identify hard computational problems, in particular, the **P**-complete languages that are believed hard to parallelize and the **NP**-complete languages that are believed hard to solve on serial computers. After more than fifty years of research it is still unknown whether **NP**-complete problems have polynomial-time algorithms.

In this chapter not only do we describe circuits for important functions, but we show that most Boolean functions are complex. For example, we show that there are so many Boolean functions on n variables and so few circuits containing C or fewer gates that unless C is large, not all Boolean functions can be realized with C gates or fewer.

Circuit complexity is also explored in Chapter 9. The present chapter develops methods to derive lower bounds on the size and depth of circuits. A lower bound on the circuit size (depth) of a function f is a value for the size (depth) below which there does not exist a circuit for f. Thus, every circuit for f must have a size (depth) greater than or equal to the lower bound. In Chapter 9 we also establish a connection between circuit depth and formula size, the number of Boolean operations needed to realize a Boolean function by a formula. This allows us to derive an upper bound on formula size from an upper bound on depth. Thus, the depth bounds of this chapter are useful in deriving upper bounds on the size of the smallest formulas for problems. Prefix circuits are used in the present chapter to design fast adders. They are also used in Chapter 6 to design fast parallel algorithms.

2.2 Straight-Line Programs and Circuits

As suggested in Section 1.4.1, the mapping between inputs and outputs of a logic circuit can be described by a binary function. In this section we formalize this idea and, in addition, demonstrate that every binary function can be realized by a circuit. Normal-form expansions of Boolean functions play a central role in establishing the latter result. Circuits were defined informally in Section 1.4.1. We now give a formal definition of circuits.

To fix ideas, we start with an example. Figure 2.1 shows a circuit that contains two AND gates, one OR gate, and two NOT gates. (Circles denote NOT gates, AND and OR gates are labeled $\wedge$ and $\vee$, respectively.) Corresponding to this circuit is the following functional de-

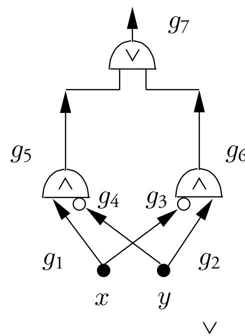


Figure 2.1 A circuit is the graph of a Boolean straight-line program.

scription of the circuit, where g_j is the value computed by the jth input or gate of the circuit:

$$\begin{array}{llllll} g_1 & := & x; & g_5 & := & g_1 \wedge g_4; \\ g_2 & := & y; & g_6 & := & g_3 \wedge g_2; \\ g_3 & := & \overline{g}_1; & g_7 & := & g_5 \vee g_6; \\ g_4 & := & \overline{g}_2; & & & \end{array} \tag{2.1}$$

The statement $g_1 := x$; means that the external input x is the value associated with the first vertex of the circuit. The statement $g_3 := \overline{g}_1$; means that the value computed at the third vertex is the NOT of the value computed at the first vertex. The statement $g_5 := g_1 \wedge g_4$; means that the value computed at the fifth vertex is the AND of the values computed at the first and fourth vertices. The statement $g_7 := g_5 \vee g_6$; means that the value computed at the seventh vertex is the OR of the values computed at the fifth and sixth vertices. The above is a description of the functions computed by the circuit. It does not explicitly specify which function(s) are the outputs of the circuit.

Shown below is an alternative description of the above circuit that contains the same information. It is a **straight-line program** whose syntax is closer to that of standard programming languages. Each step is numbered and its associated purpose is given. **Input** and **output steps** are identified by the keywords READ and OUTPUT, respectively. **Computation steps** are identified by the keywords AND, OR, and NOT.

$$\begin{array}{llll} (1 & \text{READ} & x) & \\ (2 & \text{READ} & y) & \\ (3 & \text{NOT} & 1) & \\ (4 & \text{NOT} & 2) & \\ (5 & \text{AND} & 1 & 4) \end{array} \qquad \begin{array}{llll} (6 & \text{AND} & 3 & 2) \\ (7 & \text{OR} & 5 & 6) \\ (8 & \text{OUTPUT} & 5) & \\ (9 & \text{OUTPUT} & 7) & \end{array} \tag{2.2}$$

The correspondence between the steps of a straight-line program and the functions computed at them is evident.

Straight-line programs are not limited to describing logic circuits. They can also be used to describe algebraic computations. (See Chapter 6.) In this case, a computation step is identified with a keyword describing the particular algebraic operation to be performed. In the case of

logic circuits, the operations can include many functions other than the basic three mentioned above.

As illustrated above, a straight-line program can be constructed for any circuit. Similarly, given a straight-line program, a circuit can be drawn for it as well. We now formally define straight-line programs, circuits, and characteristics of the two.

DEFINITION 2.2.1 *A* **straight-line program** *is set of steps each of which is an* **input step**, *denoted* $(s \text{ READ } x)$, *an* **output step**, *denoted* $(s \text{ OUTPUT } i)$, *or a* **computation step**, *denoted* $(s \text{ OP } i \ \dots \ k)$. *Here s is the number of a step, x denotes an input variable, and the keywords* READ, OUTPUT, *and* OP *identify steps in which an input is read, an output produced, and the operation* OP *is performed. In the sth computation step the arguments to* OP *are the results produced at steps $i, \dots, k$. It is required that these steps precede the sth step; that is, $s \geq i, \dots, k$.*

A **circuit** *is the graph of a straight-line program. (The requirement that each computation step operate on results produced in preceding steps insures that this graph is a DAG.) The* **fan-in** *of the circuit is the maximum in-degree of any vertex. The* **fan-out** *of the circuit is the maximum outdegree of any vertex. A* **gate** *is the vertex associated with a computation step.*

The **basis** Ω *of a circuit and its corresponding straight-line program is the set of operations that they use. The bases of Boolean straight-line programs and logic circuits contain only Boolean functions. The* **standard basis**, Ω_0, *for a logic circuit is the set* {AND, OR, NOT}.

2.2.1 Functions Computed by Circuits

As stated above, each step of a straight-line program computes a function. We now define the functions computed by straight-line programs, using the example given in Eq. (2.2).

DEFINITION 2.2.2 *Let g_s be the* **function computed by the sth step of a straight-line program**. *If the sth step is the input step $(s \text{ READ } x)$, then $g_s = x$. If it is the computation step $(s \text{ OP } i \ \dots \ k)$, the function is $g_s = \text{OP}(g_i, \dots, g_k)$, where $g_i, \dots, g_k$ are the functions computed at steps on which the sth step depends. If a straight-line program has n inputs and m outputs, it computes a function $f : \mathcal{B}^n \mapsto \mathcal{B}^m$. If $s_1, s_2, \dots, s_m$ are the output steps, then $f = (g_{s_1}, g_{s_2}, \dots, g_{s_m})$. The function computed by a circuit is the function computed by the corresponding straight-line program.*

The functions computed by the logic circuit of Fig. 2.1 are given below. The expression for g_s is found by substituting for its arguments the expressions derived at the steps on which it depends.

$$
\begin{array}{lllllll}
g_1 & := & x; & \qquad & g_5 & := & x \wedge \overline{y}; \\
g_2 & := & y; & & g_6 & := & \overline{x} \wedge y; \\
g_3 & := & \overline{x}; & & g_7 & := & (x \wedge \overline{y}) \vee (\overline{x} \wedge y); \\
g_4 & := & \overline{y}; & & & &
\end{array}
$$

The function computed by the above Boolean straight-line program is $f(x, y) = (g_5, g_7)$. The table of values assumed by f as the inputs x and y run through all possible values is shown below. The value of g_7 is the EXCLUSIVE OR function.

x	y	g_5	g_7
0	0	0	0
0	1	0	1
1	0	1	1
1	1	0	0

We now ask the following question: "Given a circuit with values for its inputs, how can the values of its outputs be computed?" One response it to build a circuit of physical gates, supply values for the inputs, and then wait for the signals to propagate through the gates and arrive at the outputs. A second response is to write a program in a high-level programming language to compute the values of the outputs. A simple program for this purpose assigns each step to an entry of an array and then evaluates the steps in order. This program solves the **circuit value problem**; that is, it determines the value of a circuit.

2.2.2 Circuits That Compute Functions

Now that we know how to compute the function defined by a circuit and its corresponding straight-line program, we ask: given a function, how can we construct a circuit (and straight-line program) that will compute it? Since we presume that computational tasks are defined by functions, it is important to know how to build simple machines, circuits, that will solve these tasks. In Chapter 3 we show that circuits play a central role in the design of machines with memory. Thus, whether a function or task is to be solved with a machine without memory (a circuit) or a machine with memory (such as the random-access machine), the circuit and its associated straight-line program play a key role.

To construct a circuit for a function, we begin by describing the function in a table. As seen earlier, the table for a function $f^{(n,m)} : \mathcal{B}^n \mapsto \mathcal{B}^m$ has n columns containing all 2^n possible values for the n input variables of the function. Thus, it has 2^n rows. It also has m columns containing the m outputs associated with each pattern of n inputs. If we let $x_1, x_2, \ldots, x_n$ be the **input variables** of f and let $y_1, y_2, \ldots, y_m$ be its **output variables**,

			$f^{(3,2)}_{\text{example}}$	
x_1	x_2	x_3	y_1	y_2
0	0	0	1	1
0	0	1	0	1
0	1	0	1	0
0	1	1	0	1
1	0	0	1	1
1	0	1	1	0
1	1	0	0	1
1	1	1	1	1

Figure 2.2 The truth table for the function $f^{(3,2)}_{\text{example}}$.

then we write $f(x_1, x_2, \dots, x_n) = (y_1, y_2, \dots, y_m)$. This is illustrated by the function $f^{(3,2)}_{\text{example}}(x_1, x_2, x_3) = (y_1, y_2)$ defined in Fig. 2.2 on page 39.

A **binary function** is one whose domain and codomain are Cartesian products of $\mathcal{B} = \{0, 1\}$. A **Boolean function** is a binary function whose codomain consists of the set $\mathcal{B}$. In other words, it has one output.

As we see in Section 2.3, normal forms provide standard ways to construct circuits for Boolean functions. Because a normal-form expansion of a function generally does not yield a circuit of smallest size or depth, methods are needed to simplify the algebraic expressions produced by these normal forms. This topic is discussed in Section 2.2.4.

Before exploring the algebraic properties of simple Boolean functions, we define the basic circuit complexity measures used in this book.

2.2.3 Circuit Complexity Measures

We often ask for the smallest or most shallow circuit for a function. If we need to compute a function with a circuit, as is done in central processing units, then knowing the size of the smallest circuit is important. Also important is the depth of the circuit. It takes time for signals applied to the circuit inputs to propagate to the outputs, and the length of the longest path through the circuit determines this time. When central processing units must be fast, minimizing circuit depth becomes important.

As indicated in Section 1.5, the size of a circuit also provides a lower bound on the space-time product needed to solve a problem on the random-access machine, a model for modern computers. Consequently, if the size of the smallest circuit for a function is large, its space-time product must be large. Thus, a problem can be shown to be hard to compute by a machine with memory if it can be shown that every circuit for it is large.

We now define two important circuit complexity measures.

DEFINITION 2.2.3 *The* **size** *of a logic circuit is the number of gates it contains. Its* **depth** *is the number of gates on the longest path through the circuit. The* **circuit size***, $C_\Omega(f)$, and* **circuit depth***, $D_\Omega(f)$, of a Boolean function $f : \mathcal{B}^n \mapsto \mathcal{B}^m$ are defined as the smallest size and smallest depth of any circuit, respectively, over the basis Ω for f.*

Most Boolean functions on n variables are very complex. As shown in Sections 2.12 and 2.13, their circuit size is proportional to $2^n/n$ and their depth is approximately n. Fortunately, most functions of interest have much smaller size and depth. (It should be noted that the circuit of smallest size for a function may be different from that of smallest depth.)

2.2.4 Algebraic Properties of Boolean Functions

Since the operations AND ($\wedge$), OR ($\vee$), EXCLUSIVE OR ($\oplus$), and NOT ($\neg$ or $\overline{}$) play a vital role in the construction of normal forms, we simplify the subsequent discussion by describing their properties.

If we interchange the two arguments of AND, OR, or EXCLUSIVE OR, it follows from their definition that their values do not change. This property, called **commutativity**, holds for all three operators, as stated next.

COMMUTATIVITY

$x_1 \vee x_2 = x_2 \vee x_1$
$x_1 \wedge x_2 = x_2 \wedge x_1$
$x_1 \oplus x_2 = x_2 \oplus x_1$

When constants are substituted for one of the variables of these three operators, the expression computed is simplified, as shown below.

SUBSTITUTION OF CONSTANTS

$x_1 \vee 0 = x_1$	$x_1 \wedge 1 = x_1$
$x_1 \vee 1 = 1$	$x_1 \oplus 0 = x_1$
$x_1 \wedge 0 = 0$	$x_1 \oplus 1 = \overline{x}_1$

Also, when one of the variables of one of these functions is replaced by itself or its negation, the functions simplify, as shown below.

ABSORPTION RULES

$x_1 \vee x_1 = x_1$	$x_1 \wedge x_1 = x_1$
$x_1 \vee \overline{x}_1 = 1$	$x_1 \wedge \overline{x}_1 = 0$
$x_1 \oplus x_1 = 0$	$x_1 \vee (x_1 \wedge x_2) = x_1$
$x_1 \oplus \overline{x}_1 = 1$	$x_1 \wedge (x_1 \vee x_2) = x_1$

To prove each of these results, it suffices to test exhaustively each of the values of the arguments of these functions and show that the right- and left-hand sides have the same value.

DeMorgan's rules, shown below, are very important in proving properties about circuits because they allow each AND gate to be replaced by an OR gate and three NOT gates and vice versa. The rules can be shown correct by constructing tables for each of the given functions.

DEMORGAN'S RULES

$\overline{(x_1 \vee x_2)} = \overline{x}_1 \wedge \overline{x}_2$
$\overline{(x_1 \wedge x_2)} = \overline{x}_1 \vee \overline{x}_2$

The functions AND, OR, and EXCLUSIVE OR are all **associative**; that is, all ways of combining three or more variables with any of these functions give the same result. (An operator $\odot$ is **associative** if for all values of a, b, and c, $a \odot (b \odot c) = (a \odot b) \odot c$.) Again, proof by enumeration suffices to establish the following results.

ASSOCIATIVITY

$x_1 \vee (x_2 \vee x_3) = (x_1 \vee x_2) \vee x_3$
$x_1 \wedge (x_2 \wedge x_3) = (x_1 \wedge x_2) \wedge x_3$
$x_1 \oplus (x_2 \oplus x_3) = (x_1 \oplus x_2) \oplus x_3$

Because of associativity it is not necessary to parenthesize repeated uses of the operators $\vee$, $\wedge$, and $\oplus$.

Finally, the following **distributive laws** are important in simplifying Boolean algebraic expressions. The first two laws are the same as the distributivity of integer multiplication over integer addition when multiplication and addition are replaced by AND and OR.

DISTRIBUTIVITY

$x_1 \wedge (x_2 \vee x_3) = (x_1 \wedge x_2) \vee (x_1 \wedge x_3)$
$x_1 \wedge (x_2 \oplus x_3) = (x_1 \wedge x_2) \oplus (x_1 \wedge x_3)$
$x_1 \vee (x_2 \wedge x_3) = (x_1 \vee x_2) \wedge (x_1 \vee x_3)$

We often write $x \wedge y$ as xy. The operator $\wedge$ has precedence over the operators $\vee$ and $\oplus$, which means that parentheses in $(x \wedge y) \vee z$ and $(x \wedge y) \oplus z$ may be dropped.

The above rules are illustrated by the following formula:

$$\begin{aligned}
\overline{(\overline{x} \wedge (y \oplus z))} \wedge (x \vee y) &= (x \vee \overline{(y \oplus z)}) \wedge (x \vee y) \\
&= (x \vee (\overline{y} \oplus z)) \wedge (x \vee y) \\
&= x \vee (y \wedge (\overline{y} \oplus z)) \\
&= x \vee ((y \wedge \overline{y}) \oplus (y \wedge z)) \\
&= x \vee (0 \oplus y \wedge z) \\
&= x \vee (y \wedge z)
\end{aligned}$$

DeMorgan's second rule is used to simplify the first term in the first equation. The last rule on substitution of constants is used twice to simplify the second equation. The third distributivity rule and commutativity of $\wedge$ are used to simplify the third one. The second distributivity rule is used to expand the fourth equation. The fifth equation is simplified by invoking the third absorption rule. The final equation results from the commutativity of $\oplus$ and application of the rule $x_1 \oplus 0 = x_1$. When there is no loss of clarity, we drop the operator symbol $\wedge$ between two literals.

2.3 Normal-Form Expansions of Boolean Functions

Normal forms are standard ways of constructing circuits from the tables defining Boolean functions. They are easy to apply, although the circuits they produce are generally far from optimal. They demonstrate that every Boolean function can be realized over the standard basis as well as the basis containing AND and EXCLUSIVE OR.

In this section we define five normal forms: the disjunctive and conjunctive normal forms, the sum-of-products expansion, the product-of-sums expansion, and the ring-sum expansion.

2.3.1 Disjunctive Normal Form

A **minterm** in the variables $x_1, x_2, \dots, x_n$ is the AND of each variable or its negation. For example, when $n = 3$, $\overline{x}_1 \wedge \overline{x}_2 \wedge \overline{x}_3$ is a minterm. It has value 1 exactly when each variable has value 0. $x_1 \wedge \overline{x}_2 \wedge x_3$ is another minterm; it has value 1 exactly when $x_1 = 1$, $x_2 = 0$ and $x_3 = 1$. It follows that a minterm on n variables has value 1 for exactly one of the 2^n points in its domain. Using the notation $x^1 = x$ and $x^0 = \overline{x}$, we see that the above minterms can be written as $x_1^0 x_2^0 x_3^0$ and $x_1 x_2^0 x_3$, respectively, when we drop the use of the AND operator $\wedge$. Thus, $x_1^0 x_2^0 x_3^0 = 1$ when $\boldsymbol{x} = (x_1, x_2, x_3) = (0, 0, 0)$ and $x_1^1 x_2^0 x_3^1 = 1$ when $\boldsymbol{x} = (1, 0, 1)$. That is, the minterm $\boldsymbol{x}^{(\boldsymbol{c})} = x_1^{c_1} \wedge x_2^{c_2} \wedge \cdots \wedge x_n^{c_n}$ has value 1 exactly when $\boldsymbol{x} = \boldsymbol{c}$ where $\boldsymbol{c} = (c_1, c_2, \dots, c_n)$. A **minterm of a Boolean function** f is a minterm $\boldsymbol{x}^{(\boldsymbol{c})}$ that contains all the variables of f and for which $f(\boldsymbol{c}) = 1$.

The word "disjunction" is a synonym for OR, and the **disjunctive normal form (DNF)** of a Boolean function $f : \mathcal{B}^n \mapsto \mathcal{B}$ is the OR of the minterms of f. Thus, f has value 1 when

x_1	x_2	x_3	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

(a)

x_1	x_2	x_3	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

(b)

Figure 2.3 Truth tables illustrating the disjunctive and conjunctive normal forms.

exactly one of its minterms has value 1 and has value 0 otherwise. Consider the function whose table is given in Fig. 2.3(a). Its disjunctive normal form (or minterm expansion) is given by the following formula:

$$f(x_1, x_2, x_3) = x_1^0 x_2^0 x_3^0 \vee x_1^0 x_2^1 x_3^0 \vee x_1^1 x_2^0 x_3^0 \vee x_1^1 x_2^0 x_3^1 \vee x_1^1 x_2^1 x_3^1$$

The **parity** function $f_{\oplus}^{(n)} : \mathcal{B}^n \mapsto \mathcal{B}$ on n inputs has value 1 when an odd number of inputs is 1 and value 0 otherwise. It can be realized by a circuit containing $n-1$ instances of the EXCLUSIVE OR operator; that is, $f_{\oplus}^{(n)}(x_1, \ldots, x_n) = x_1 \oplus x_2 \oplus \cdots \oplus x_n$. However, the DNF of $f_{\oplus}^{(n)}$ contains 2^{n-1} minterms, a number exponential in n. The DNF of $f_{\oplus}^{(3)}$ is

$$f_{\oplus}^{(3)}(x, y, z) = \overline{x}\,\overline{y}\,z \ \vee \ \overline{x}\,y\,\overline{z} \ \vee \ x\,\overline{y}\,\overline{z} \ \vee \ xyz$$

Here we use the standard notation for a variable and its complement.

2.3.2 Conjunctive Normal Form

A **maxterm** in the variables $x_1, x_2, \ldots, x_n$ is the OR of each variable or its negation. For example, $x_1 \vee x_2 \vee \overline{x}_3$ is a maxterm. It has value 0 exactly when $x_1 = x_2 = 0$ and $x_3 = 1$. $x_1 \vee \overline{x}_2 \vee \overline{x}_3$ is another maxterm; it has value 0 exactly when $x_1 = 0$ and $x_2 = x_3 = 1$. It follows that a maxterm on n variables has value 0 for exactly one of the 2^n points in its domain. We see that the above maxterms can be written as $x_1^1 \vee x_2^1 \vee x_3^0$ and $x_1^1 \vee x_2^0 \vee x_3^0$, respectively. Thus, $x_1^1 \vee x_2^1 \vee x_3^0 = 0$ when $\boldsymbol{x} = (x_1, x_2, x_3) = (0, 0, 1)$ and $x_1^1 \vee x_2^0 \vee x_3^1 = 0$ when $\boldsymbol{x} = (0, 1, 0)$. That is, the maxterm $\boldsymbol{x}^{(\boldsymbol{c})} = x_1^{\overline{c}_1} \vee x_2^{\overline{c}_2} \vee \cdots \vee x_n^{\overline{c}_n}$ has value 0 exactly when $\boldsymbol{x} = \boldsymbol{c}$. A **maxterm of a Boolean function** f is a maxterm $\boldsymbol{x}^{(\boldsymbol{c})}$ that contains all the variables of f and for which $f(\boldsymbol{c}) = 0$.

The word "conjunction" is a synonym for AND, and the **conjunctive normal form (CNF)** of a Boolean function $f : \mathcal{B}^n \mapsto \mathcal{B}$ is the AND of the maxterms of f. Thus, f has value 0 when exactly one of its maxterms has value 0 and has value 1 otherwise. Consider the function whose table is given in Fig. 2.3(b). Its conjunctive normal form (or maxterm expansion) is given by the following formula:

$$f(x_1, x_2, x_3) = (x_1^1 \vee x_2^1 \vee x_3^0) \wedge (x_1^1 \vee x_2^0 \vee x_3^0) \wedge (x_1^0 \vee x_2^0 \vee x_3^1)$$

An important relationship holds between the DNF and CNF representations for Boolean functions. If DNF(f) and CNF(f) are the representations of f in the DNF and CNF expansions, then the following identity holds (see Problem 2.6):

$$\mathrm{CNF}(f) = \overline{\mathrm{DNF}(\overline{f})}$$

It follows that the CNF of the parity function $f_{\oplus}^{(n)}$ has 2^{n-1} maxterms.

Since each function $f : \mathcal{B}^n \mapsto \mathcal{B}^m$ can be expanded to its CNF or DNF and each can be realized with circuits, the following result is immediate.

THEOREM 2.3.1 *Every function $f : \mathcal{B}^n \mapsto \mathcal{B}^m$ can be realized by a logic circuit.*

2.3.3 SOPE and POSE Normal Forms

The sum-of-products and product-of-sums normal forms are simplifications of the disjunctive and conjunctive normal forms, respectively. These simplifications are obtained by using the rules stated in Section 2.2.4.

A **product** in the variables $x_{i_1}, x_{i_2}, \dots, x_{i_k}$ is the AND of each of these variables or their negations. For example, $x_2\, \overline{x}_5\, x_6$ is a product. A minterm is a product that contains each of the variables of a function. A **product of a Boolean function** f is a product in some of the variables of f. A **sum-of-products expansion (SOPE)** of a Boolean function is the OR (the sum) of products of f. Thus, the DNF is a special case of the SOPE of a function.

A SOPE of a Boolean function can be obtained by simplifying the DNF of a function using the rules given in Section 2.2.4. For example, the DNF given earlier and shown below can be simplified to produce a SOPE.

$$y_1(x_1, x_2, x_3) = \overline{x}_1\, \overline{x}_2\, \overline{x}_3 \vee \overline{x}_1\, x_2\, \overline{x}_3 \vee x_1\, \overline{x}_2\, \overline{x}_3 \vee x_1\, \overline{x}_2\, x_3 \vee x_1\, x_2\, x_3$$

It is easy to see that the first and second terms combine to give $\overline{x}_1 \overline{x}_3$, the first and third give $\overline{x}_2 \overline{x}_3$ (we use the property that $g \vee g = g$), and the last two give $x_1 x_3$. That is, we can write the following SOPE for f:

$$f = x_1\, x_3 \vee \overline{x}_1\, \overline{x}_3 \vee \overline{x}_2 \overline{x}_3 \qquad (2.3)$$

Clearly, we could have stopped before any one of the above simplifications was used and generated another SOPE for f. This illustrates the point that a Boolean function may have many SOPEs but only one DNF.

A **sum** in the variables $x_{i_1}, x_{i_2}, \dots, x_{i_k}$ is the OR of each of these variables or their negations. For example, $\overline{x}_3 \vee x_4 \vee x_7$ is a sum. A maxterm is a product that contains each of the variables of a function. A **sum of a Boolean function** f is a sum in some of the variables of f. A **product-of-sum expansion (POSE)** of a Boolean function is the AND (the product) of sums of f. Thus, the CNF is a special case of the POSE of a function.

A POSE of a Boolean function can be obtained by simplifying the CNF of a function using the rules given in Section 2.2.4. For example, the conjunction of the two maxterms $x_1 \vee \overline{x}_2 \vee \overline{x}_3$ and $x_1 \vee \overline{x}_2 \vee x_3$, namely $(x_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (x_1 \vee \overline{x}_2 \vee x_3)$, can be reduced to $x_1 \vee \overline{x}_2$ by the application of rules of Section 2.2.4, as shown below:

$$(x_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (x_1 \vee \overline{x}_2 \vee x_3) =$$

$$= x_1 \vee (\overline{x}_2 \vee \overline{x}_3) \wedge (\overline{x}_2 \vee x_3) \quad \{\text{3rd distributivity rule}\}$$
$$= x_1 \vee \overline{x}_2 \vee (\overline{x}_3 \wedge x_3) \quad \{\text{3rd distributivity rule}\}$$
$$= x_1 \vee \overline{x}_2 \vee 0 \quad \{\text{6th absorption rule}\}$$
$$= x_1 \vee \overline{x}_2 \quad \{\text{1st rule on substitution of constants}\}$$

It is easily shown that the POSE of the parity function is its CNF. (See Problem 2.8.)

2.3.4 Ring-Sum Expansion

The **ring-sum expansion (RSE)** of a function f is the EXCLUSIVE OR ($\oplus$) of a constant and products ($\wedge$) of unnegated variables of f. For example, $1 \oplus x_1x_3 \oplus x_2x_4$ is an RSE. The operations $\oplus$ and $\wedge$ over the set $\mathcal{B} = \{0, 1\}$ constitute a ring. (Rings are examined in Section 6.2.1.) Any two instances of the same product in the RSE can be eliminated since they sum to 0.

The RSE of a Boolean function $f : \mathcal{B}^n \mapsto \mathcal{B}$ can be constructed from its DNF, as we show. Since a minterm of f has value 1 on exactly one of the 2^n points in its domain, at most one minterm in the DNF for f has value 1 for any point in its domain. Thus, we can combine minterms with EXCLUSIVE OR instead of OR without changing the value of the function. Now replace $\overline{x}_i$ with $x_i \oplus 1$ in each minterm containing $\overline{x}_i$ and then apply the second distributivity rule. We simplify the resulting formula by using commutativity and the absorption rule $x_i \oplus x_i = 0$. For example, since the minterms of $(\overline{x}_1 \vee x_2)x_3$ are $\overline{x}_1x_2x_3$, $\overline{x}_1\overline{x}_2x_3$, and $x_1x_2x_3$, we construct the RSE of this function as follows:

$$\begin{aligned}
(\overline{x}_1 \vee x_2)x_3 &= \overline{x}_1x_2x_3 \oplus \overline{x}_1\overline{x}_2x_3 \oplus x_1x_2x_3 \\
&= (x_1 \oplus 1)x_2x_3 \oplus (x_1 \oplus 1)(x_2 \oplus 1)x_3 \oplus x_1x_2x_3 \\
&= x_2x_3 \oplus x_1x_2x_3 \oplus x_3 \oplus x_1x_3 \oplus x_2x_3 \oplus x_1x_2x_3 \oplus x_1x_2x_3 \\
&= x_3 \oplus x_1x_3 \oplus x_1x_2x_3
\end{aligned}$$

The third equation follows by applying the second distributivity rule and commutativity. The fourth follows by applying $x_i \oplus x_i = 0$ and commutativity. The two occurrences of x_2x_3 are canceled, as are two of the three instances of $x_1x_2x_3$.

As this example illustrates, the RSE of a function $f : \mathcal{B}^n \mapsto \mathcal{B}$ is the EXCLUSIVE OR of a Boolean constant c_0 and one or more products of unnegated variables of f. Since each of the n variables of f can be present or absent from a product, there are 2^n products, including the product that contains no variables; that is, a constant whose value is 0 or 1. For example, $1 \oplus x_3 \oplus x_1x_3 \oplus x_1x_2x_3$ is the RSE of the function $\overline{(\overline{x}_1 \vee x_2)\, x_3}$.

2.3.5 Comparison of Normal Forms

It is easy to show that the RSE of a Boolean function is unique (see Problem 2.7). However, the RSE is not necessarily a compact representation of a function. For example, the RSE of the OR of n variables, $f_{\vee}^{(n)}$, includes every product term except for the constant 1. (See Problem 2.9.)

It is also true that some functions have large size in some normal forms but small size in others. For example, the parity function has exponential size in the DNF and CNF normal forms but linear size in the RSE. Also, $f_{\vee}^{(n)}$ has exponential size in the RSE but linear size in the CNF and SOPE representations.

A natural question to ask is whether there is a function that has large size in all five normal forms. The answer is yes. This is true of the Boolean function on n variables whose value is 1 when the sum of its variables is 0 modulo 3 and is 0 otherwise. It has exponential-size DNF, CNF, and RSE normal forms. (See Problem 2.10.) However, its smallest circuit is linear in n. (See Section 2.11.)

2.4 Reductions Between Functions

A common way to solve a new problem is to apply an existing solution to it. For example, an integer multiplication algorithm can be used to square an integer by supplying two copies of the integer to the multiplier. This idea is called a "reduction" in complexity theory because we reduce one problem to a previously solved problem, here squaring to integer multiplication. In this section we briefly discuss several simple forms of reduction, including subfunctions. Note that the definitions given below are not limited to binary functions.

DEFINITION 2.4.1 *A function $f : \mathcal{A}^n \mapsto \mathcal{A}^m$ is a* **reduction** *to the function $g : \mathcal{A}^r \mapsto \mathcal{A}^s$ through application of the functions $p : \mathcal{A}^s \mapsto \mathcal{A}^m$ and $q : \mathcal{A}^n \mapsto \mathcal{A}^r$ if for all $\boldsymbol{x} \in \mathcal{A}^n$:*

$$f(\boldsymbol{x}) = p(g(q(\boldsymbol{x})))$$

As suggested in Fig. 2.4, it follows that circuits for q, g and p can be cascaded (the output of one is the input to the next) to form a circuit for f. Thus, the circuit size and depth of f, $C(f)$ and $D(f)$, satisfy the following inequalities:

$$C(f) \leq C(p) + C(g) + C(q)$$
$$D(f) \leq D(p) + D(g) + D(q)$$

A special case of a reduction is the subfunction, as defined below.

DEFINITION 2.4.2 *Let $g : \mathcal{A}^n \mapsto \mathcal{A}^m$. A* **subfunction** *$f$ of g is a function obtained by assigning values to some of the input variables of g, assigning (not necessarily unique) variable names to the rest, deleting and/or permuting some of its output variables. We say that f is a* **reduction** *to g via the* **subfunction relationship**.

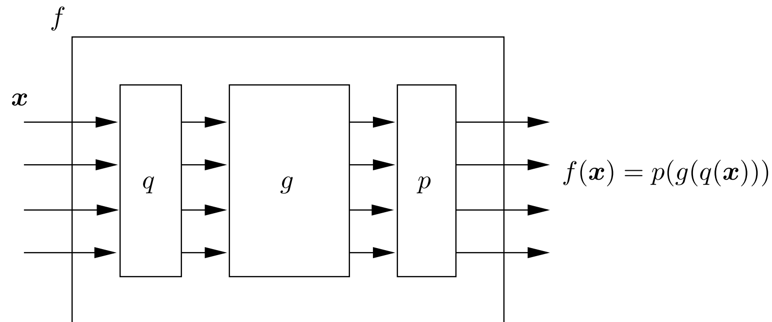


Figure 2.4 The function f is reduced to the function g by applying functions p and q to prepare the input to g and manipulate its output.

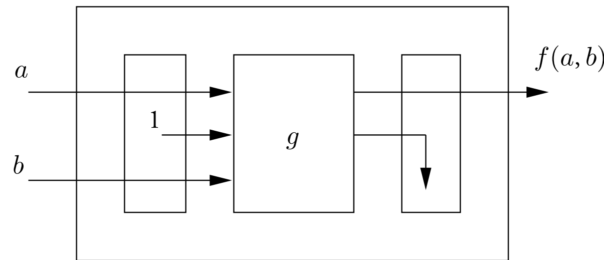


Figure 2.5 The subfunction f of the function g is obtained by fixing some input variables, assigning names to the rest, and deleting and/or permuting outputs.

This definition is illustrated by the function $f^{(3,2)}_{\text{example}}(x_1, x_2, x_3) = (y_1, y_2)$ in Fig. 2.2. We form the subfunction y_1 by deleting y_2 from $f^{(3,2)}_{\text{example}}$ and fixing $x_1 = a$, $x_2 = 1$, and $x_3 = b$, where a and b are new variables. Then, consulting (2.3), we see that y_1 can be written as follows:

$$\begin{aligned} y_1 &= (a\, b) \vee (\overline{a}\, \overline{b}) \vee (\overline{1}\, \overline{b}) \\ &= a\, b \vee \overline{a}\, \overline{b} \\ &= a \oplus b \oplus 1 \end{aligned}$$

That is, y_1 contains the complement of the EXCLUSIVE OR function as a subfunction. The definition is also illustrated by the reductions developed in Sections 2.5.2, 2.5.6, 2.9.5, and 2.10.1.

The subfunction definition derives its importance from the following lemma. (See Fig. 2.5.)

LEMMA 2.4.1 *If f is a subfunction of g, a straight-line program for f can be created from one for g without increasing the size or depth of its circuit.*

As shown in Section 2.9.5, the logical shifting function (Section 2.5.1) can be realized by composing the integer multiplication and decoder functions (Section 2.5). This type of reduction is useful in those cases in which one function is reduced to another with the aid of functions whose complexity (size or depth or both) is known to be small relative to that of either function. It follows that the two functions have the same asymptotic complexity even if we cannot determine what that complexity is. The reduction is a powerful idea that is widely used in computer science. Not only is it the essence of the subroutine, but it is also used to classify problems by their time or space complexity. (See Sections 3.9.3 and 8.7.)

2.5 Specialized Circuits

A small number of special functions arise repeatedly in the design of computers. These include logical and shifting operations, encoders, decoders, multiplexers, and demultiplexers. In the following sections we construct efficient circuits for these functions.

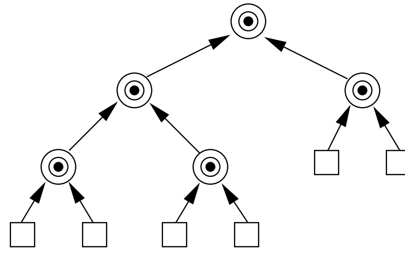


Figure 2.6 A balanced binary tree circuit that combines elements with an associative operator.

2.5.1 Logical Operations

Logical operations are not only building blocks for more complex operations, but they are at the heart of all central processing units. Logical operations include "vector" and "associating" operations. A **vector operation** is the component-wise operation on one or more vectors. For example, the vector NOT on the vector $\boldsymbol{x} = (x_{n-1}, \ldots, x_1, x_0)$ is the vector $\overline{\boldsymbol{x}} = (\overline{x}_{n-1}, \ldots, \overline{x}_1, \overline{x}_0)$. Other vector operations involve the application of a two-input function to corresponding components of two vectors. If $\star$ is a two-input function, such as AND or OR, and $\boldsymbol{x} = (x_{n-1}, \ldots, x_1, x_0)$ and $\boldsymbol{y} = (y_{n-1}, \ldots, y_1, y_0)$ are two n-tuples, the vector operation $\boldsymbol{x} \star \boldsymbol{y}$ is

$$\boldsymbol{x} \star \boldsymbol{y} = (x_{n-1} \star y_{n-1}, \ldots, x_1 \star y_1, x_0 \star y_0)$$

An **associative operator** $\odot$ over a $\mathcal{A}$ satisfies the condition $(a \odot b) \odot c = a \odot (b \odot c)$ for all $a, b, c \in \mathcal{A}$. A **summing operation** on an n-tuple $\boldsymbol{x}$ with an associative two-input operation $\odot$ produces the "sum" y defined below.

$$y = x_{n-1} \odot \cdots \odot x_1 \odot x_0$$

An efficient circuit for computing y is shown in Fig. 2.6. It is a binary tree whose leaves are associated with the variables $x_{n-1}, \ldots, x_1, x_0$. Each level of the tree is full except possibly the last. This circuit has smallest depth of those that form the associative combination of the variables, namely $\lceil \log_2 n \rceil$.

2.5.2 Shifting Functions

Shifting functions can be used to multiply integers and generally manipulate data. A cyclic shifting function rotates the bits in a word. For example, the left cyclic shift of the 4-tuple $(1, 0, 0, 0)$ by three places produces the 4-tuple $(0, 1, 0, 0)$.

The **cyclic shifting function** $f^{(n)}_{\text{cyclic}} : \mathcal{B}^{n+\lceil \log_2 n \rceil} \mapsto \mathcal{B}^n$ takes as input an n-tuple $\boldsymbol{x} = (x_{n-1}, \ldots, x_1, x_0)$ and cyclically shifts it left by $|\boldsymbol{s}|$ places, where $|\boldsymbol{s}|$ is the integer associated with the binary k-tuple $\boldsymbol{s} = (s_{k-1}, \ldots, s_1, s_0)$, $k = \lceil \log_2 n \rceil$, and

$$|\boldsymbol{s}| = \sum_{j=0}^{k-1} s_j 2^j$$

The n-tuple that results from the shift is $\boldsymbol{y} = (y_{n-1}, \ldots, y_1, y_0)$, denoted as follows:

$$\boldsymbol{y} = f^{(n)}_{\text{cyclic}}(\boldsymbol{x}, \boldsymbol{s})$$

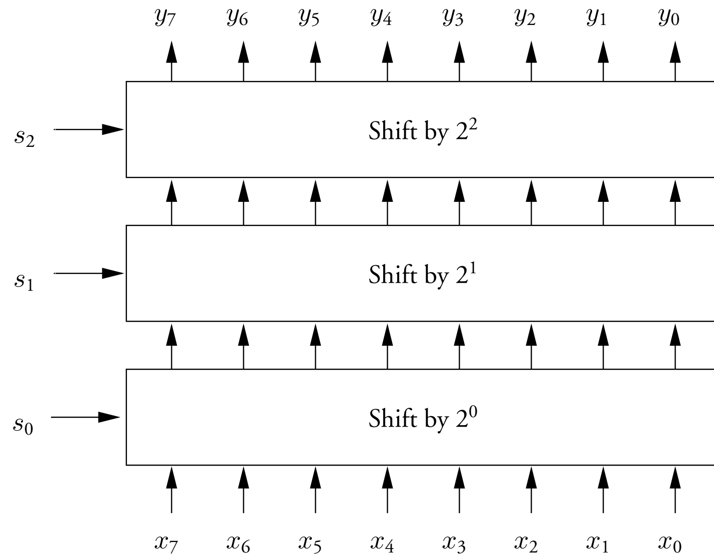


Figure 2.7 Three stages of a cyclic shifting circuit on eight inputs.

A convenient way to perform the cyclic shift of $\boldsymbol{x}$ by $|\boldsymbol{s}|$ places is to represent $|\boldsymbol{s}|$ as a sum of powers of 2, as shown above, and for each $0 \leq j \leq k-1$, shift $\boldsymbol{x}$ left cyclically by $s_j 2^j$ places, that is, by either 0 or 2^j places depending on whether $s_j = 0$ or 1. For example, consider cyclically shifting the 8-tuple $\boldsymbol{u} = (1,0,1,1,0,1,0,1)$ by seven places. Since 7 is represented by the binary number $(1,1,1)$, that is, $7 = 4+2+1$, to shift $(1,0,1,1,0,1,0,1)$ by seven places it suffices to shift it by one place, by two places, and then by four places. (See Fig. 2.7.)

For $0 \leq r \leq n-1$, the following formula defines the value of the rth output, y_r, of a circuit on n inputs that shifts its input $\boldsymbol{x}$ left cyclically by either 0 or 2^j places depending on whether $s_j = 0$ or 1:

$$y_r = (x_r \wedge \overline{s}_j) \vee (x_{(r-2^j) \bmod n} \wedge s_j)$$

Thus, y_r is x_r in the first case or $x_{(r-2^j) \bmod n}$ in the second. The subscript $(r-2^j) \bmod n$ is the positive remainder of $(r-2^j)$ after division by n. For example, if $n=4$, $r=1$, and $j=1$, then $(r-2^j) = -1$, which is 3 modulo 4. That is, in a circuit that shifts by either 0 or 2^1 places, y_1 is either x_1 or x_3 because x_3 moves into the second position when shifted left cyclically by two places.

A circuit based on the above formula that shifts by either 0 or 2^j places depending on whether $s_j = 0$ or 1 is shown in Fig. 2.8 for $n=4$. The circuit on n inputs has $3n+1$ gates and depth 3.

It follows that a circuit for cyclic shifting an n-tuple can be realized in $k = \lceil \log_2 n \rceil$ stages each of which has $3n+1$ gates and depth 3, as suggested by Fig. 2.7. Since this may be neither the smallest nor the shallowest circuit that computes $f^{(n)}_{\text{cyclic}} : \mathcal{B}^{n+\lceil \log_2 n \rceil}$, its minimal circuit size and depth satisfy the following bounds.

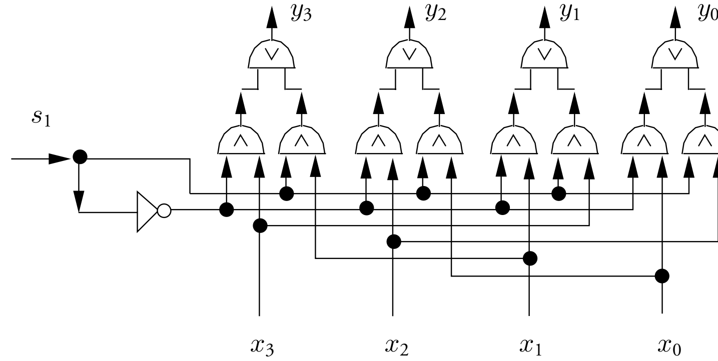


Figure 2.8 One stage of a circuit for cyclic shifting four inputs by 0 or 2 places depending on whether $s_1 = 0$ or 1.

LEMMA 2.5.1 *The cyclic shifting function* $f^{(n)}_{\text{cyclic}} : \mathcal{B}^{n+\lceil \log_2 n \rceil} \mapsto \mathcal{B}^n$ *can be realized by a circuit of the following size and depth over the basis* $\Omega_0 = \{\wedge, \vee, \neg\}$*:*

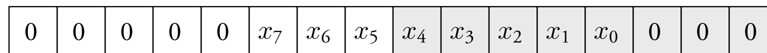
$$C_{\Omega_0}\left(f^{(n)}_{\text{cyclic}}\right) \leq (3n+1)\lceil \log_2 n \rceil$$
$$D_{\Omega_0}\left(f^{(n)}_{\text{cyclic}}\right) \leq 3\lceil \log_2 n \rceil$$

The **logical shifting function** $f^{(n)}_{\text{shift}} : \mathcal{B}^{n+\lceil \log_2 n \rceil} \mapsto \mathcal{B}^n$ shifts left the n-tuple $\boldsymbol{x}$ by a number of places specified by a binary $\lceil \log n \rceil$-tuple $\boldsymbol{s}$, discarding the higher-index components, and filling in the lower-indexed vacated places with 0's to produce the n-tuple $\boldsymbol{y}$, where

$$y_j = \begin{cases} x_{j-|\boldsymbol{s}|} & \text{for } |\boldsymbol{s}| \leq j \leq n-1 \\ 0 & \text{otherwise} \end{cases}$$

REDUCTIONS BETWEEN LOGICAL AND CYCLIC SHIFTING The logical shifting function $f^{(n)}_{\text{shift}} : \mathcal{B}^{n+\lceil \log_2 n \rceil} \mapsto \mathcal{B}^n$ on the n-tuple $\boldsymbol{x}$ is defined below in terms of $f^{(2n)}_{\text{cyclic}}$ and the "projection" function $\pi^{(n)}_L : \mathcal{B}^{2n} \mapsto \mathcal{B}^n$ that deletes the n high order components from its input $2n$-tuple. Here $\mathbf{0}$ denotes the zero binary n-tuple and $\mathbf{0} \cdot \boldsymbol{x}$ denotes the concatenation of the two strings. (See Figs. 2.9 and 2.10.)

$$f^{(n)}_{\text{shift}}(\boldsymbol{x}, \boldsymbol{s}) = \pi^{(n)}_L \left(f^{(2n)}_{\text{cyclic}}(\mathbf{0} \cdot \boldsymbol{x}, \boldsymbol{s})\right)$$



0	0	0	0	0	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	0	0	0

Figure 2.9 The reduction of $f^{(8)}_{\text{shift}}$ to $f^{(8)}_{\text{cyclic}}$ obtained by cyclically shifting $\mathbf{0} \cdot \boldsymbol{x}$ by three places and projecting out the shaded components.

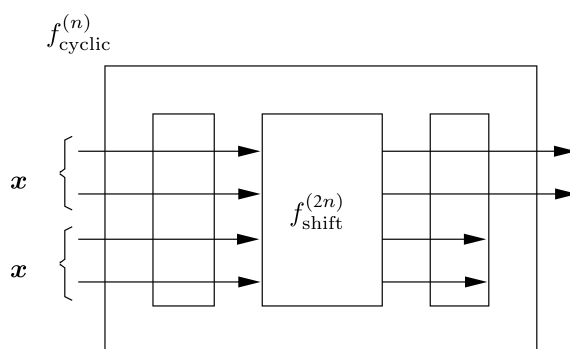


Figure 2.10 The function $f^{(n)}_{\text{cyclic}}$ is obtained by computing $f^{(2n)}_{\text{shift}}$ on $\boldsymbol{x}\boldsymbol{x}$ and truncating the n low-order bits.

LEMMA 2.5.2 *The function $f^{(2n)}_{\text{cyclic}}$ contains $f^{(n)}_{\text{shift}}$ as a subfunction and the function $f^{(2n)}_{\text{shift}}$ contains $f^{(n)}_{\text{cyclic}}$ as a subfunction.*

Proof The first statement follows from the above argument concerning $f^{(n)}_{\text{shift}}$. The second statement follows by noting that

$$f^{(n)}_{\text{cyclic}}(\boldsymbol{x}, \boldsymbol{s}) = \pi^{(n)}_H \left(f^{(2n)}_{\text{shift}}(\boldsymbol{x} \cdot \boldsymbol{x}, \boldsymbol{s}) \right)$$

where $\pi^{(n)}_H$ deletes the n low-order components of its input. ■

This relationship between logical and cyclic shifting functions clearly holds for variants of such functions in which the amount of a shift is specified with some other notation. An example of such a shifting function is integer multiplication in which one of the two arguments is a power of 2.

2.5.3 Encoder

The **encoder function** $f^{(n)}_{\text{encode}} : \mathcal{B}^{2^n} \mapsto \mathcal{B}^n$ has 2^n inputs, exactly one of which is 1. Its output is an n-tuple that is a binary number representing the position of the input that has value 1. That is, it encodes the position of the input bit that has value 1. Encoders are used in CPUs to identify the source of external interrupts.

Let $\boldsymbol{x} = (x_{2^n-1}, \dots, x_2, x_1, x_0)$ represent the 2^n inputs and let $\boldsymbol{y} = (y_{n-1}, \dots, y_1, y_0)$ represent the n outputs. Then, we write $f^{(n)}_{\text{encode}}(\boldsymbol{x}) = \boldsymbol{y}$.

When $n = 1$, the encoder function has two inputs, x_1 and x_0, and one output, y_0, whose value is $y_0 = x_1$ because if $x_0 = 1$, then $x_1 = 0$ and $y_0 = 0$ is the binary representation of the input whose value is 1. Similar reasoning applies when $x_0 = 0$.

When $n \geq 2$, we observe that the high-order output bit, y_{n-1}, has value 1 if 1 falls among the variables $x_{2^n-1}, \dots, x_{2^{n-1}+1}, x_{2^{n-1}}$. Otherwise, $y_{n-1} = 0$. Thus, y_{n-1} can be computed as the OR of these variables, as suggested for the encoder on eight inputs in Fig. 2.11.

The remaining $n - 1$ output bits, $y_{n-2}, \dots, y_1, y_0$, represent the position of the 1 among variables $x_{2^{n-1}-1}, \dots, x_2, x_1, x_0$ if $y_{n-1} = 0$ or the 1 among variables $x_{2^n-1}, \dots, x_{2^{n-1}+1}$, $x_{2^{n-1}}$ if $y_{n-1} = 1$. For example, for $n = 3$ if $\boldsymbol{x} = (0, 0, 0, 0, 0, 0, 1, 0)$, then $y_2 = 0$ and

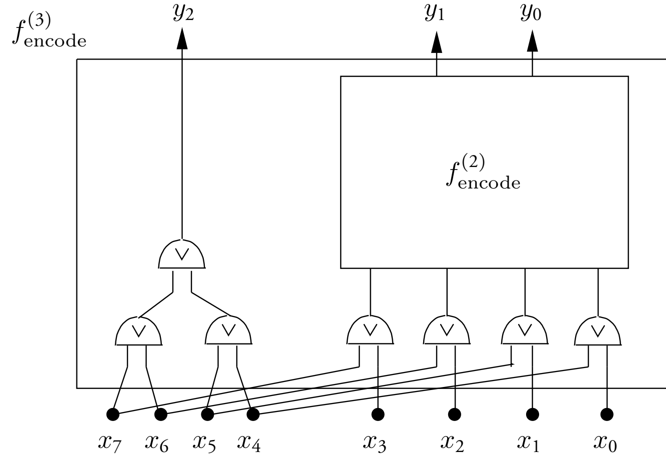


Figure 2.11 The recursive construction of an encoder circuit on eight inputs.

$(y_1, y_0) = (0, 1)$, whereas if $\boldsymbol{x} = (0, 0, 1, 0, 0, 0, 0, 0)$, then $y_2 = 1$ and $(y_1, y_0) = (0, 1)$. Thus, after computing y_{n-1} as the OR of the 2^{n-1} high-order inputs, the remaining output bits can be obtained by supplying to an encoder on 2^{n-1} inputs the 2^{n-1} low-order bits if $y_{n-1} = 0$ or the 2^{n-1} high-order bits if $y_{n-1} = 1$. It follows that in both cases we can supply the vector $\boldsymbol{\delta} = (x_{2^n-1} \vee x_{2^{(n-1)}-1}, x_{2^n-2} \vee x_{2^{(n-1)}-2}, \dots, x_{2^{(n-1)}} \vee x_0)$ of $2^{(n-1)}$ components to the encoder on $2^{(n-1)}$ inputs. This is illustrated in Fig. 2.11.

Let's now derive upper bounds on the size and depth of the optimal circuit for $f^{(n)}_{\text{encode}}$. Clearly $C_{\Omega_0}\left(f^{(1)}_{\text{encode}}\right) = 0$ and $D_{\Omega_0}\left(f^{(1)}_{\text{encode}}\right) = 0$, since no gates are needed in this case. From the construction described above and illustrated in Fig. 2.11, we see that we can construct a circuit for $f^{(n)}_{\text{encode}}$ in a two-step process. First, we form y_{n-1} as the OR of the 2^{n-1} high-order variables in a balanced OR tree of depth n using $2^{n-1} - 1$ OR's. Second, we form the vector $\boldsymbol{\delta}$ with a circuit of depth 1 using 2^{n-1} OR's and supply it to a copy of a circuit for $f^{(n-1)}_{\text{encode}}$. This provides the following recurrences for the circuit size and depth of $f^{(n)}_{\text{encode}}$ because the depth of this circuit is no more than the maximum of the depth of the OR tree and 1 more than the depth of a circuit for $f^{(n-1)}_{\text{encode}}$:

$$C_{\Omega_0}\left(f^{(n)}_{\text{encode}}\right) \leq 2^n - 1 + C_{\Omega_0}(f^{(n-1)}_{\text{encode}}) \tag{2.4}$$

$$D_{\Omega_0}\left(f^{(n)}_{\text{encode}}\right) \leq \max(n - 1, D_{\Omega_0}(f^{(n-1)}_{\text{encode}}) + 1) \tag{2.5}$$

The solutions to these recurrences are stated as the following lemma, as the reader can show. (See Problem 2.14.)

LEMMA 2.5.3 *The encoder function $f^{(n)}_{\text{encode}}$ has the following circuit size and depth bounds:*

$$C_{\Omega_0}\left(f^{(n)}_{\text{encode}}\right) \leq 2^{n+1} - (n + 3)$$

$$D_{\Omega_0}\left(f^{(n)}_{\text{encode}}\right) \leq n - 1$$

2.5.4 Decoder

A decoder is a function that reverses the operation of an encoder: given an n-bit binary address, it generates 2^n bits with a single 1 in the position specified by the binary number. Decoders are used in the design of random-access memory units (see Section 3.5) and of the multiplexer (see Section 2.5.5).

The **decoder function** $f_{\text{decode}}^{(n)} : \mathcal{B}^n \mapsto \mathcal{B}^{2^n}$ has n input variables $\boldsymbol{x} = (x_{n-1}, \dots, x_1, x_0)$ and 2^n output variables $\boldsymbol{y} = (y_{2^n-1}, \dots, y_1, y_0)$; that is, $f_{\text{decode}}^{(n)}(\boldsymbol{x}) = \boldsymbol{y}$. Let $\boldsymbol{c}$ be a binary n-tuple corresponding to the integer $|\boldsymbol{c}|$. All components of the binary 2^n-tuple $\boldsymbol{y}$ are zero except for the one whose index is $|\boldsymbol{c}|$, namely $y_{|\boldsymbol{c}|}$. Thus, the minterm functions in the variables $\boldsymbol{x}$ are computed as the output of $f_{\text{decode}}^{(n)}$.

A direct realization of the function $f_{\text{decode}}^{(n)}$ can be obtained by realizing each minterm independently. This circuit uses $(2n-1)2^n$ gates and has depth $\lceil \log_2 n \rceil + 1$. Thus we have the following bounds over the basis $\Omega_0 = \{\wedge, \vee, \neg\}$:

$$C_{\Omega_0}\left(f_{\text{decode}}^{(n)}\right) \le (2n-1)2^n$$

$$D_{\Omega_0}\left(f_{\text{decode}}^{(n)}\right) \le \lceil \log_2 n \rceil + 1$$

A smaller upper bound on circuit size and depth can be obtained from the recursive construction of Fig. 2.12, which is based on the observation that a minterm on n variables is the AND of a minterm on the first $n/2$ variables and a minterm on the second $n/2$ variables. For example, when $n = 4$, the minterm $\overline{x}_3 \wedge x_2 \wedge \overline{x}_1 \wedge \overline{x}_0$ is obviously equal to the AND of the minterm $\overline{x}_3 \wedge x_2$ in the variables x_3 and x_2 and the minterm $\overline{x}_1 \wedge \overline{x}_0$ in the variables x_1 and x_0. Thus, when n is even, the minterms that are the outputs of $f_{\text{decode}}^{(n)}$ can be formed by ANDing

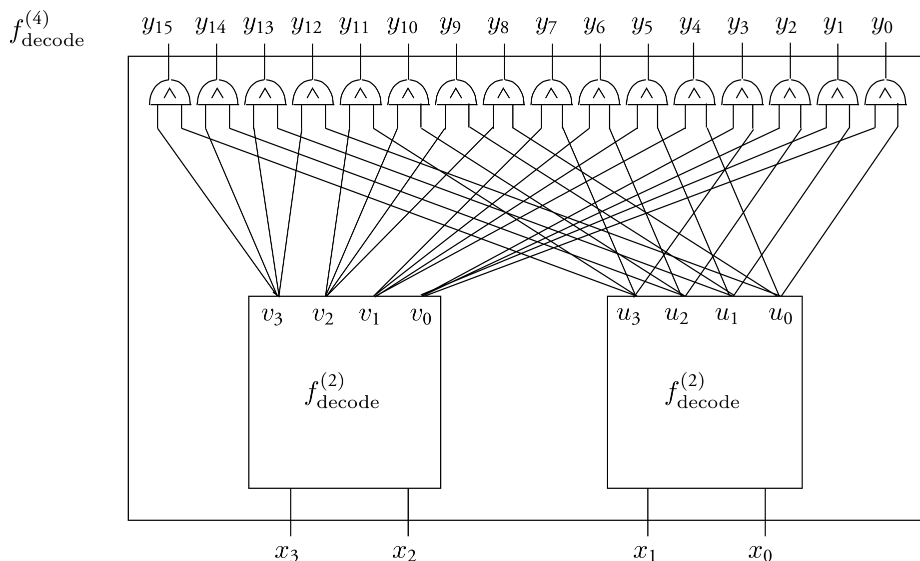


Figure 2.12 The construction of a decoder on four inputs from two copies of a decoder on two inputs.

every minterm generated by a circuit for $f_{\mathrm{decode}}^{(n/2)}$ on the variables $x_{n/2-1}, \ldots, x_0$ with every minterm generated by a circuit for $f_{\mathrm{decode}}^{(n/2)}$ on the variables $x_{n-1}, \ldots, x_{n/2}$, as suggested in Fig. 2.12.

The new circuit for $f_{\mathrm{decode}}^{(n)}$ has a size that is at most twice that of a circuit for $f_{\mathrm{decode}}^{(n/2)}$ plus 2^n for the AND gates that combine minterms. It has a depth that is at most 1 more than the depth of a circuit for $f_{\mathrm{decode}}^{(n/2)}$. Thus, when n is even we have the following bounds on the circuit size and depth of $f_{\mathrm{decode}}^{(n)}$:

$$\begin{aligned} C_{\Omega_0}\left(f_{\mathrm{decode}}^{(n)}\right) &\leq 2C_{\Omega_0}\left(f_{\mathrm{decode}}^{(n/2)}\right) + 2^n \\ D_{\Omega_0}\left(f_{\mathrm{decode}}^{(n)}\right) &\leq D_{\Omega_0}\left(f_{decode}^{(n/2)}\right) + 1 \end{aligned}$$

Specializing the first bounds given above on the size and depth of a decoder circuit to one on $n/2$ inputs, we have the bound in Lemma 2.5.4. Furthermore, since the output functions are all different, $C_{\Omega_0}\left(f_{\mathrm{decode}}^{(n)}\right)$ is at least 2^n.

LEMMA 2.5.4 *For n even the decoder function $f_{\mathrm{decode}}^{(n)}$ has the following circuit size and depth bounds:*

$$\begin{aligned} 2^n \leq C_{\Omega_0}\left(f_{\mathrm{decode}}^{(n)}\right) &\leq 2^n + (2n-2)2^{n/2} \\ D_{\Omega_0}\left(f_{\mathrm{decode}}^{(n)}\right) &\leq \lceil \log_2 n \rceil + 1 \end{aligned}$$

The circuit size bound is linear in the number of outputs. Also, for $n \geq 12$, the exact value of $C_{\Omega_0}\left(f_{\mathrm{decode}}^{(n)}\right)$ is known to within 25%. Since each output depends on n inputs, we will see in Chapter 9 that the upper bound on depth is exactly the depth of the smallest depth circuit for the decoder function.

2.5.5 Multiplexer

The **multiplexer function** $f_{\mathrm{mux}}^{(n)} : \mathcal{B}^{2^n+n} \mapsto \mathcal{B}$ has two vector inputs, $\boldsymbol{z} = (z_{2^n-1}, \ldots, z_1, z_0)$ and $\boldsymbol{x} = (x_{n-1}, \ldots, x_1, x_0)$, where $\boldsymbol{x}$ is treated as an address. The output of $f_{\mathrm{mux}}^{(n)}$ is $v = z_j$, where $j = |\boldsymbol{x}|$ is the integer represented by the binary number $\boldsymbol{x}$. This function is also known as the **storage access function** because it simulates the access to storage made by a random-access memory with one-bit words. (See Section 3.5.)

The similarity between this function and the decoder function should be apparent. The decoder function has n inputs, $\boldsymbol{x} = (x_{n-1}, \ldots, x_1, x_0)$, and 2^n outputs, $\boldsymbol{y} = (y_{2^n-1}, \ldots, y_1, y_0)$, where $y_j = 1$ if $j = |\boldsymbol{x}|$ and $y_j = 0$ otherwise. Thus, we can form $v = z_j$ as

$$v = (z_{2^n-1} \wedge y_{2^n-1}) \vee \cdots \vee (z_1 \wedge y_1) \vee (z_0 \wedge y_0)$$

This circuit uses a circuit for the decoder function $f_{\mathrm{decode}}^{(n)}$ plus 2^n AND gates and $2^n - 1$ OR gates. It adds a depth of $n + 1$ to the depth of a decoder circuit. Lemma 2.5.5 follows immediately from these observations.

LEMMA 2.5.5 *The multiplexer function $f_{\text{mux}}^{(n)} : \mathcal{B}^{2^n+n} \mapsto \mathcal{B}$ can be realized with the following circuit size and depth over the basis $\Omega_0 = \{\wedge, \vee, \neg\}$:*

$$C_{\Omega_0}\left(f_{\text{mux}}^{(n)}\right) \leq 3 \cdot 2^n + 2(n-1)2^{n/2} - 1$$
$$D_{\Omega_0}\left(f_{\text{mux}}^{(n)}\right) \leq n + \lceil \log_2 n \rceil + 2$$

Using the lower bound of Theorem 9.3.3, one can show that it is impossible to reduce the upper bound on circuit size to less than $2^{n+1} - 2$. At the cost of increasing the depth by 1, the circuit size bound can be improved to about 2^{n+1}. (See Problem 2.15.) Since $f_{\text{mux}}^{(n)}$ depends on $2^n + n$ variables, we see from Theorem 9.3.1 that it must have depth at least $\log_2(2^n + n) \geq n$. Thus, the above depth bound is very tight.

2.5.6 Demultiplexer

The **demultiplexer function** $f_{\text{demux}}^{(n)} : \mathcal{B}^{n+1} \mapsto \mathcal{B}^{2^n}$ is very similar to a decoder. It has $n+1$ inputs consisting of n bits, $\boldsymbol{x}$, that serve as an address and a data bit e. It has 2^n outputs $\boldsymbol{y}$ all of which are 0 if $e = 0$ and one output that is 1 if $e = 1$, namely the output specified by the n address bits. Demultiplexers are used to route a data bit (e) to one of 2^n output positions.

A circuit for the demultiplexer function can be constructed as follows. First, form the AND of e with each of the n address bits $x_{n-1}, \dots, x_1, x_0$ and supply this new n-tuple as input to a decoder circuit. Let $\boldsymbol{z} = (z_{2^n-1}, \dots, z_1, z_0)$ be the decoder outputs. When $e = 0$, each of the decoder inputs is 0 and each of the decoder outputs except z_0 is 0 and $z_0 = 1$. If we form the AND of z_0 with e, this new output is also 0 when $e = 0$. If $e = 1$, the decoder input is the address $\boldsymbol{x}$ and the output that is 1 is in the position specified by this address. Thus, a circuit for a demultiplexer can be constructed from a circuit for $f_{\text{decode}}^{(n)}$ to which are added n AND gates on its input and one on its output. This circuit has a depth that is at most 2 more than the depth of the decoder circuit. Since a circuit for a decoder can be constructed from one for a demultiplexer by fixing $e = 1$, we have the following bounds on the size and depth of a circuit for $f_{\text{demux}}^{(n)}$.

LEMMA 2.5.6 *The demultiplexer function $f_{\text{demux}}^{(n)} : \mathcal{B}^{n+1} \mapsto \mathcal{B}^{2^n}$ can be realized with the following circuit size and depth over the basis $\Omega_0 = \{\wedge, \vee, \neg\}$:*

$$0 \leq C_{\Omega_0}\left(f_{\text{demux}}^{(n)}\right) - C_{\Omega_0}\left(f_{\text{decoder}}^{(n)}\right) \leq n + 1$$
$$0 \leq D_{\Omega_0}\left(f_{\text{demux}}^{(n)}\right) - D_{\Omega_0}\left(f_{\text{decoder}}^{(n)}\right) \leq 2$$

2.6 Prefix Computations

The prefix computation first appeared in the design of logic circuits, the goal being to parallelize as much as possible circuits for integer addition and multiplication. The carry-lookahead adder is a fast circuit for integer addition that is based on a prefix computation. (See Section 2.7.) Prefix computations are now widely used in parallel computation because they provide a standard, optimizable framework in which to perform computations in parallel.

The **prefix function** $\mathcal{P}_{\odot}^{(n)} : \mathcal{A}^n \mapsto \mathcal{A}^n$ on input $\boldsymbol{x} = (x_1, x_2, \dots, x_n)$ produces as output $\boldsymbol{y} = (y_1, y_2, \dots, y_n)$, which is a running sum of its n inputs $\boldsymbol{x}$ using the operator

$\odot$ as the summing operator. That is, $y_j = x_1 \odot x_2 \odot \cdots \odot x_j$ for $1 \leq j \leq n$. Thus, if the set $\mathcal{A}$ is $\mathbb{N}$, the natural numbers, and $\odot$ is the integer addition operator $+$, then $\mathcal{P}_+^{(n)}$ on the input $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ produces the output $\boldsymbol{y}$, where $y_1 = x_1$, $y_2 = x_1 + x_2$, $y_3 = x_1 + x_2 + x_3, \ldots, y_n = x_1 + x_2 + \cdots + x_n$. For example, shown below is the prefix function on a 6-vector of integers under integer addition.

$$\boldsymbol{x} = (2, 1, 3, 7, 5, 1)$$
$$\mathcal{P}_+^{(6)}(\boldsymbol{x}) = (2, 3, 6, 13, 18, 19)$$

A prefix function is defined only for operators $\odot$ that are associative over the set $\mathcal{A}$. An **operator over $\mathcal{A}$ is associative** if a) for all a and b in $\mathcal{A}$, $a \odot b$ is in $\mathcal{A}$, and b) for all a, b, and c in $\mathcal{A}$, $(a \odot b) \odot c = a \odot (b \odot c)$—that is, if all groupings of terms in a sum with the operator $\odot$ have the same value. A pair $(\mathcal{A}, \odot)$ in which $\odot$ is associative is called a **semigroup**. Three semigroups on which a prefix function can be defined are

- $(\mathbb{N}, +)$ where $\mathbb{N}$ are the natural numbers and $+$ is integer addition.
- $(\{0, 1\}^*, \cdot)$ where $\{0, 1\}^*$ is the set of binary strings and $\cdot$ is string concatenation.
- $(\mathcal{A}, \odot_{\text{copy}})$ where $\mathcal{A}$ is a set and $\odot_{\text{copy}}$ is defined by $a \odot_{\text{copy}} b = a$.

It is easy to show that the concatenation operator $\cdot$ on $\{0, 1\}^*$ and $\odot_{\text{copy}}$ on a set $\mathcal{A}$ are associative. (See Problem 2.20.) Another important semigroup is the set of matrices under matrix multiplication (see Theorem 6.2.1).

Summarizing, if $(\mathcal{A}, \odot)$ is a semigroup, the prefix function $\mathcal{P}_\odot^{(n)} : \mathcal{A}^n \mapsto \mathcal{A}^n$ on input $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ produces as output $\boldsymbol{y} = (y_1, y_2, \ldots, y_n)$, where $y_j = x_1 \odot x_2 \odot \cdots \odot x_j$ for $1 \leq j \leq n$.

Load balancing on a parallel machine is an important application of prefix computation. A simple example of load balancing is the following: We assume that p processors, numbered from 0 to $p - 1$, are running processes in parallel. We also assume that processes are born and die, resulting in a possible imbalance in the number of processes active on processors. Since it is desirable that all processors be running the same number of processes, processes are periodically redistributed among processors to balance the load. To rebalance the load, a) processors are given a linear order, and b) each process is assigned a Boolean variable with value 1 if it is alive and 0 otherwise. Each processor computes its number of living processes, n_i. A prefix computation is then done on these values using the linear order among processors. This computation provides the jth processor with the sum $n_j + n_{j-1} + \cdots + n_1$ which it uses to give each of its living processes a unique index. The sum $n = n_p + \cdots + n_1$ is then broadcast to all processors. When the processors are in balance all have $\lceil n/p \rceil$ processes except possibly one that has fewer processes. Assigning the sth process to processor $(s \bmod p)$ insures that the load is balanced.

Another important type of prefix computation is the **segmented prefix computation**. In this case two n-vectors are given, a **value vector $\boldsymbol{x}$** and a **flag vector $\boldsymbol{\phi}$**. The value of the ith entry y_i in the result vector $\boldsymbol{y}$ is x_i if ϕ_i is 1 and otherwise is the associative combination with $\odot$ of x_i and the values between it and the first value x_j to the left of x_i for which the flag $\phi_j = 1$. The first bit of ϕ is always 1. An example of a segmented prefix computation is shown below for integer values and integer addition as the associative operation:

$$\boldsymbol{x} = (2, 1, 3, 7, 5, 1)$$

$$\phi = (1,0,0,1,0,1)$$
$$\boldsymbol{y} = (2,3,6,7,12,1)$$

As shown in Problem 2.21, a segmented prefix computation is a special case of a general prefix computation. This is demonstrated by defining a new associative operation $\otimes$ on value-flag pairs that returns another value-flag pair.

2.6.1 An Efficient Parallel Prefix Circuit

A circuit for the prefix function $\mathcal{P}_{\odot}^{(n)}$ can be realized with $O(n^2)$ instances of $\odot$ if for each $1 \leq j \leq n$ we naively realize $y_j = x_1 \odot x_2 \odot \cdots \odot x_j$ with a separate circuit containing $j-1$ instances of $\odot$. If each such circuit is organized as a balanced binary tree, the depth of the circuit for $\mathcal{P}_{\odot}^{(n)}$ is the depth of the circuit for y_n, which is $\lceil \log_2 n \rceil$. This is a parallel circuit for the prefix problem but uses many more operators than necessary. We now describe a much more efficient circuit for this problem; it uses $O(n)$ instances of $\odot$ and has depth $O(\log n)$.

To describe this improved circuit, we let $x[r,r] = x_r$ and for $r \leq s$ let $x[r,s] = x_r \odot x_{r+1} \odot \cdots \odot x_s$. Then we can write $\mathcal{P}_{\odot}^{(n)}(\boldsymbol{x}) = \boldsymbol{y}$ where $y_j = x[1,j]$.

Because $\odot$ is associative, we observe that $x[r,s] = x[r,t] \odot x[t+1,s]$ for $r \leq t < s$. We use this fact to construct the improved circuit. Let $n = 2^k$. Observe that if we form the $(n/2)$-tuple $(x[1,2], x[3,4], x[5,6], \ldots, x[2^k-1, 2^k])$ using the rule $x[i,i+1] = x[i,i] \odot x[i+1,i+1]$ for i odd and then do a prefix computation on it, we obtain the $(n/2)$-tuple $(x[1,2], x[1,4], x[1,6], \ldots, x[1,2^k])$. This is almost what is needed. We must only compute $x[1,1]$, $x[1,3]$, $x[1,5]$, $\ldots$, $x[1,2^k-1]$, which is easily done using the rule $x[1,2i+1] = x[1,2i] \odot x_{2i+1}$ for $1 \leq i \leq 2^{k-1}-1$. (See Fig. 2.13.) The base case for this construction is that of $n = 1$, for which $y_1 = x_1$ and no operations are needed.

If $C(k)$ is the size of this circuit on $n = 2^k$ inputs and $D(k)$ is its depth, then $C(0) = 0$, $D(0) = 0$ and $C(k)$ and $D(k)$ for $k \geq 1$ satisfy the following recurrences:

$$C(k) = C(k-1) + 2^k - 1$$
$$D(k) = D(k-1) + 2$$

As a consequence, we have the following result.

THEOREM 2.6.1 *For $n = 2^k$, k an integer, the parallel prefix function $\mathcal{P}_{\odot}^{(n)} : \mathcal{A}^n \mapsto \mathcal{A}^n$ on an n-vector with associative operator $\odot$ can be implemented by a circuit with the following size and depth bounds over the basis $\Omega = \{\odot\}$:*

$$C_{\Omega}\left(\mathcal{P}_{\odot}^{(n)}\right) \leq 2n - \log_2 n - 2$$
$$D_{\Omega}\left(\mathcal{P}_{\odot}^{(n)}\right) \leq 2\log_2 n$$

Proof The solution to the recurrence on $C(k)$ is $C(k) = 2^{k+1} - k - 2$, as the reader can easily show. It satisfies the base case of $k = 0$ and the general case as well. The solution to $D(k)$ is $D(k) = 2k$. ■

When n is not a power of 2, we can start with a circuit for the next higher power of 2 and then delete operations and edges that are not used to produce the first n outputs.

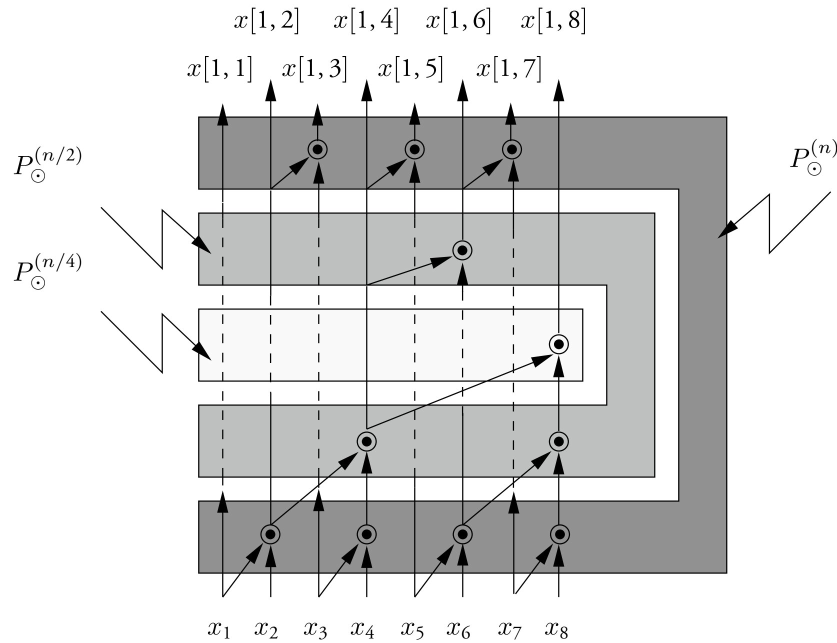


Figure 2.13 A simple recursive construction of a prefix circuit when $n = 2^k = 8$. The gates used at each stage of the construction are grouped into individual shaded regions.

2.7 Addition

Addition is a central operation in all general-purpose digital computers. In this section we describe the standard ripple adder and the fast carry-lookahead addition circuits. The ripple adder mimics the elementary method of addition taught to beginners but for binary instead of decimal numbers. Carry-lookahead addition is a fast addition method based on the fast prefix circuit described in the preceding section.

Consider the binary representation of integers in the set $\{0, 1, 2, \dots, 2^n - 1\}$. They are represented by binary n-tuples $\boldsymbol{u} = (u_{n-1}, u_{n-2}, \dots, u_1, u_0)$ and have value

$$|\boldsymbol{u}| = \sum_{j=0}^{n-1} u_j 2^j$$

where $\sum$ denotes integer addition.

The **addition function** $f_{\text{add}}^{(n)} : \mathcal{B}^{2n} \mapsto \mathcal{B}^{n+1}$ computes the sum of two binary n-bit numbers $\boldsymbol{u}$ and $\boldsymbol{v}$, as shown below, where $+$ denotes integer addition:

$$|\boldsymbol{u}| + |\boldsymbol{v}| = \sum_{j=0}^{n-1} (u_j + v_j) 2^j$$

The tuple $((u_{n-1}+v_{n-1}), (u_{n-2}+v_{n-2}), \dots, (u_0+v_0))$ is not a binary number because the coefficients of the powers of 2 are not Boolean. However, if the integer $u_0 + v_0$ is converted to

a binary number (c_1, s_0), where $c_1 2^1 + s_0 2^0 = u_0 + v_0$, then the sum can be replaced by

$$|\boldsymbol{u}| + |\boldsymbol{v}| = \sum_{j=2}^{n-1} (u_j + v_j) 2^j + (u_1 + v_1 + c_1) 2^1 + s_0 2^0$$

where the least significant bit is now Boolean. In turn, the sum $u_1 + v_1 + c_1$ can be represented in binary by (c_2, s_1), where $c_2 2 + s_1 = u_1 + v_1 + c_1$. The sum $|\boldsymbol{u}| + |\boldsymbol{v}|$ can then be replaced by one in which the two least significant coefficients are Boolean. Repeating this process on all coefficients, we have the **ripple adder** shown in Fig. 2.14.

In the general case, the jth stage of a ripple adder combines the jth coefficients of each binary number, namely u_j and v_j, and the carry from the previous stage, c_j, and represents their integer sum with the binary notation (c_{j+1}, s_j), where

$$c_{j+1} 2 + s_j = u_j + v_j + c_j$$

Here c_{j+1}, the number of 2's in the sum $u_j + v_j + c_j$, is the carry into the $(j + 1)$st stage and s_j, the number of 1's in the sum modulo 2, is the external output from the jth stage. The circuit performing this mapping is called a **full adder** (see Fig. 2.15). As the reader can easily show by constructing a table, this circuit computes the function $f_{\mathrm{FA}} : \mathcal{B}^3 \mapsto \mathcal{B}^2$, where $f_{\mathrm{FA}}(u_j, v_j, c_j) = (c_{j+1}, s_j)$ is described by the following formulas:

$$\begin{aligned} p_j &= u_j \oplus v_j \\ g_j &= u_j \wedge v_j \\ c_{j+1} &= (p_j \wedge c_j) \vee g_j \\ s_j &= p_j \oplus c_j \end{aligned} \tag{2.6}$$

Here p_j and g_j are intermediate variables with a special significance. If $g_j = 1$, a carry is **generated** at the jth stage. If $p_j = 1$, a carry from the previous stage is **propagated** through the jth stage, that is, a carry-out occurs exactly when a carry-in occurs. Note that p_j and g_j cannot both have value 1.

The full adder can be realized with five gates and depth 3. Since the first full adder has value 0 for its carry input, three gates can be eliminated from its circuit and its depth reduced by 2. It follows that a ripple adder can be realized by a circuit with the following size and depth.

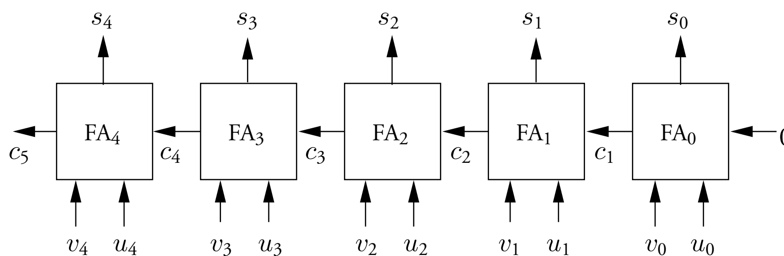


Figure 2.14 A ripple adder for binary numbers.

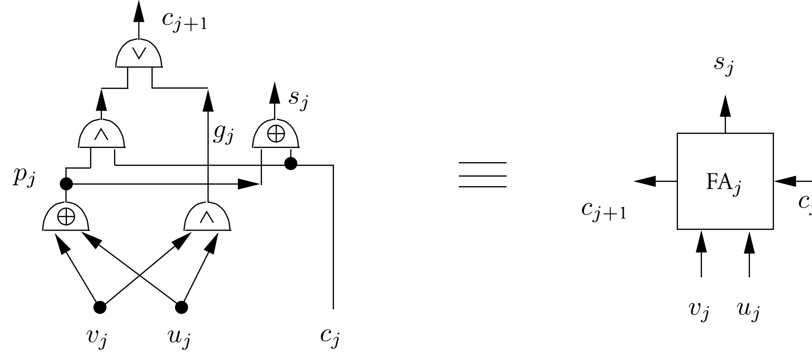


Figure 2.15 A full adder realized with gates.

THEOREM 2.7.1 *The addition function $f_{\text{add}}^{(n)} : \mathcal{B}^{2n} \mapsto \mathcal{B}^{n+1}$ can be realized with a ripple adder with the following size and depth bounds over the basis $\Omega = \{\wedge, \vee, \oplus\}$:*

$$C_\Omega\left(f_{\text{add}}^{(n)}\right) \leq 5n - 3$$
$$D_\Omega\left(f_{\text{add}}^{(n)}\right) \leq 3n - 2$$

(Do the ripple adders actually have depth less than $3n - 2$?)

2.7.1 Carry-Lookahead Addition

The ripple adder is economical; it uses a small number of gates. Unfortunately, it is slow. The depth of the circuit, a measure of its speed, is linear in n, the number of bits in each integer. The carry-lookahead adder described below is considerably faster. It uses the parallel prefix circuit described in the preceding section.

The **carry-lookahead adder** circuit is obtained by applying the prefix operation to pairs in $\mathcal{B}^2$ using the associative operator $\diamond : (\mathcal{B}^2)^2 \mapsto \mathcal{B}^2$ defined below. Let (a, b) and (c, d) be arbitrary pairs in $\mathcal{B}^2$. Then $\diamond$ is defined by the following formula:

$$(a, b) \diamond (c, d) = (a \wedge c,\ (b \wedge c) \vee d)$$

To show that $\diamond$ is associative, it suffices to show by straightforward algebraic manipulation that for all values of a, b, c, d, e, and f the following holds:

$$\begin{aligned}((a,\ b) \diamond (c,\ d)) \diamond (e,\ f) &= (a,\ b) \diamond ((c,\ d) \diamond (e,\ f)) \\ &= (a\,c\,e,\ b\,c\,e \vee d\,e \vee f)\end{aligned}$$

Let $\pi[j, j] = (p_j, g_j)$ and, for $j < k$, let $\pi[j, k] = \pi[j, k-1] \diamond \pi[k, k]$. By induction it is straightforward to show that the first component of $\pi[j, k]$ is 1 if and only if a carry propagates through the full adder stages numbered $j, j+1, \ldots, k$ and its second component is 1 if and only if a carry is generated at the rth stage, $j \leq r \leq k$, and propagates from that stage through the kth stage. (See Problem 2.26.)

The prefix computation on the string $(\pi[0, 0], \pi[1, 1], \ldots, \pi[n-1, n-1])$ with the operator $\diamond$ produces the string $(\pi[0, 0], \pi[0, 1], \pi[0, 2], \ldots, \pi[0, n-1])$. The first component of

$\pi[0, j]$ is 1 if and only if a carry generated at the zeroth stage, c_0, is propagated through the jth stage. Since $c_0 = 0$, this component is not used. The second component of $\pi[0, j]$, c_{j+1}, is 1 if and only if a carry is generated at or before the jth stage. From (2.6) we see that the sum bit generated at the jth stage, s_j, satisfies $s_j = p_j \oplus c_j$. Thus the jth output bit, s_j, is obtained from the EXCLUSIVE OR of p_j and the second component of $\pi[0, j - 1]$.

THEOREM 2.7.2 *For $n = 2^k$, k an integer, the addition function $f_{\text{add}}^{(n)} : \mathcal{B}^{2n} \mapsto \mathcal{B}^{n+1}$ can be realized with a carry-lookahead adder with the following size and depth bounds over the basis $\Omega = \{\wedge, \vee, \oplus\}$:*

$$C_\Omega\left(f_{\text{add}}^{(n)}\right) \leq 8n$$

$$D_\Omega\left(f_{\text{add}}^{(n)}\right) \leq 4\log_2 n + 2$$

Proof The prefix circuit uses $2n - \log_2 n - 3$ instances of $\diamond$ and has depth $2\log_2 n$. Since each instance of $\diamond$ can be realized by a circuit of size 3 and depth 2, each of these bounds is multiplied by these factors. Since the first component of $\pi[0, j]$ is not used, the propagate value computed at each output combiner vertex can be eliminated. This saves one gate per result bit, or n gates. However, for each $0 \leq j \leq n - 1$ we need two gates to compute p_j and q_j and one gate to compute s_j, $3n$ additional gates. The computation of these three sets of functions adds depth 2 to that of the prefix circuit. This gives the desired bounds. ■

The addition function $f_{\text{add}}^{(n)}$ is computed by the carry-lookahead adder circuit with 1.6 times as many gates as the ripple adder but in logarithmic instead of linear depth.

When exact addition is expected and every number is represented by n bits, a carry-out of the last stage of an adder constitutes an **overflow**, an error.

2.8 Subtraction

Subtraction is possible when negative numbers are available. There are several ways to represent negative numbers. To demonstrate that subtraction is not much harder than addition, we consider the **signed two's complement** representation for positive and negative integers in the set $\mathbb{Z}(n) = \{-2^n, \ldots, -2, -1, 0, 1, 2, \ldots, 2^n - 1\}$. Each signed number **u** is represented by an $(n + 1)$-tuple $(\sigma, \boldsymbol{u})$, where σ is its sign and $\boldsymbol{u} = (u_{n-1}, \ldots, u_0)$ is a binary number that is either the magnitude $|\mathbf{u}|$ of the number **u**, if positive, or the **two's complement** $2^n - |\mathbf{u}|$ of it, if negative. The sign σ is defined below:

$$\sigma = \begin{cases} 0 & \text{the number } \mathbf{u} \text{ is positive or zero} \\ 1 & \text{the number } \mathbf{u} \text{ is negative} \end{cases}$$

The two's complement of an n-bit binary number $\boldsymbol{v}$ is easily formed by adding 1 to $t = 2^n - 1 - |\boldsymbol{v}|$. Since $2^n - 1$ is represented as the n-tuple of 1's, t is obtained by complementing (NOTing) every bit of $\boldsymbol{v}$. Thus, the two's complement of $\boldsymbol{u}$ is obtained by complementing every bit of $\boldsymbol{u}$ and then adding 1. It follows that the two's complement of the two's complement of a number is the number itself. Thus, the magnitude of a negative number $(1, \boldsymbol{u})$ is the two's complement of $\boldsymbol{u}$.

This is illustrated by the integers in the set $\mathbb{Z}(4) = \{-16, \ldots, -2, -1, 0, 1, 2, \ldots, 15\}$. The two's complement representation of the decimal integers 9 and -11 are

$$9 = (0, 1, 0, 0, 1)$$
$$-11 = (1, 0, 1, 0, 1)$$

Note that the two's complement of 11 is $16 - 11 = 5$, which is represented by the four-tuple $(0, 1, 0, 1)$. The value of the two's complement of 11 can be computed by complementing all bits in its binary representation $(1, 0, 1, 1)$ and adding 1.

We now show that to add two numbers $\mathbf{u}$ and $\mathbf{v}$ in two's complement notation $(\sigma_u, \boldsymbol{u})$ and $(\sigma_v, \boldsymbol{v})$, we add them as binary $(n+1)$-tuples and discard the overflow bit, that is, the coefficient of 2^{n+1}. We now show that this procedure provides a correct answer when no overflow occurs and establish conditions on which overflow does occur.

Let $|\mathbf{u}|$ and $|\mathbf{v}|$ denote the magnitudes of the two numbers. There are four cases for their sum $\mathbf{u} + \mathbf{v}$:

Case	$\mathbf{u}$	$\mathbf{v}$	$\mathbf{u} + \mathbf{v}$
I	≥ 0	≥ 0	$\lvert\mathbf{u}\rvert + \lvert\mathbf{v}\rvert$
II	≥ 0	< 0	$2^{n+1} + \lvert\mathbf{u}\rvert - \lvert\mathbf{v}\rvert$
III	< 0	≥ 0	$2^{n+1} - \lvert\mathbf{u}\rvert + \lvert\mathbf{v}\rvert$
IV	< 0	< 0	$2^{n+1} + 2^{n+1} - \lvert\mathbf{u}\rvert - \lvert\mathbf{v}\rvert$

In the first case the sum is positive. If the coefficient of 2^n is 1, an overflow error is detected. In the second case, if $|\mathbf{u}| - |\mathbf{v}|$ is negative, then $2^{n+1} + |\mathbf{u}| - |\mathbf{v}| = 2^n + 2^n - ||\mathbf{u}| - |\mathbf{v}||$ and the result is in two's complement notation with sign 1, as it should be. If $|\mathbf{u}| - |\mathbf{v}|$ is positive, the coefficient of 2^n is 0 (a carry-out of the last stage has occurred) and the result is a positive number with sign bit 0, properly represented. A similar statement applies to the third case. In the fourth case, if $|\mathbf{u}| + |\mathbf{v}|$ is less than 2^n, the sum is $2^{n+1} + 2^n + (2^n - (|\mathbf{u}| + |\mathbf{v}|))$, which is $2^n + (2^n - (|\mathbf{u}| + |\mathbf{v}|))$ when the coefficient of 2^{n+1} is discarded. This is a proper representation for a negative number. However, if $|\mathbf{u}| + |\mathbf{v}| \geq 2^n$, a borrow occurs from the $(n+1)$st position and the sum $2^{n+1} + 2^n + (2^n - (|\mathbf{u}| + |\mathbf{v}|))$ has a 0 in the $(n+1)$st position, which is not a proper representation for a negative number (after discarding 2^{n+1}); overflow has occurred.

The following procedure can be used to subtract integer $\mathbf{u}$ from integer $\mathbf{v}$: form the two's complement of $\mathbf{u}$ and add it to the representation for $\mathbf{v}$. The negation of a number is obtained by complementing its sign and taking the two's complement of its binary n-tuple. It follows that subtraction can be done with a circuit of size linear in n and depth logarithmic in n. (See Problem 2.27.)

2.9 Multiplication

In this section we examine several methods of multiplying integers. We begin with the standard elementary integer multiplication method based on the binary representation of numbers. This method requires $O(n^2)$ gates and has depth $O(\log^2 n)$ on n-bit numbers. We then examine a divide-and-conquer method that has the same depth but much smaller circuit size. We also describe fast multiplication methods, that is, methods that have circuits with smaller depths. These include a circuit whose depth is much smaller than $O(\log n)$. It uses a novel

representation of numbers, namely, the exponents of numbers in their prime number decomposition.

The integer multiplication function $f^{(n)}_{\text{mult}} : \mathcal{B}^{2n} \mapsto \mathcal{B}^{2n}$ can be realized by the **standard integer multiplication algorithm**, which is based on the following representation for the product of integers represented as binary n-tuples $\boldsymbol{u}$ and $\boldsymbol{v}$:

$$|\boldsymbol{u}||\boldsymbol{v}| = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} u_i v_j 2^{i+j} \tag{2.7}$$

Here $|\boldsymbol{u}|$ and $|\boldsymbol{v}|$ are the magnitudes of the integers represented by $\boldsymbol{u}$ and $\boldsymbol{v}$. The standard algorithm forms the products $u_i v_j$ individually to create n binary numbers, as suggested below. Here each row corresponds to a different number; the columns correspond to powers of 2 with the rightmost column corresponding to the least significant component, namely the coefficient of 2^0.

2^6	2^5	2^4	2^3	2^2	2^1	2^0	
			u_0v_3	u_0v_2	u_0v_1	u_0v_0	$= \boldsymbol{z_0}$
		u_1v_3	u_1v_2	u_1v_1	u_1v_0	0	$= \boldsymbol{z_1}$
	u_2v_3	u_2v_2	u_2v_1	u_2v_0	0	0	$= \boldsymbol{z_2}$
u_3v_3	u_3v_2	u_3v_1	u_3v_0	0	0	0	$= \boldsymbol{z_3}$

(2.8)

Let the ith binary number produced by this multiplication operation be $\boldsymbol{z_i}$. Since each of these n binary numbers contains at most $2n-1$ bits, we treat them as if they were $(2n-1)$-bit numbers. If these numbers are added in the order shown in Fig. 2.16(a) using a carry-lookahead adder at each step, the time to perform the additions, measured by the depth of a circuit, is $O(n \log n)$. The size of this circuit is $O(n^2)$. A faster circuit containing about the same number of gates can be constructed by adding $\boldsymbol{z_0}, \dots, \boldsymbol{z_{n-1}}$ in a balanced binary tree with n leaves, as shown in Fig. 2.16(b). This tree has $n-1$ $(2n-1)$-bit adders. (A binary tree with n leaves has $n-1$ internal vertices.) If each of the adders is a carry-lookahead adder, the depth of this circuit is $O(\log^2 n)$ because the tree has $O(\log n)$ adders on every path from the root to a leaf.

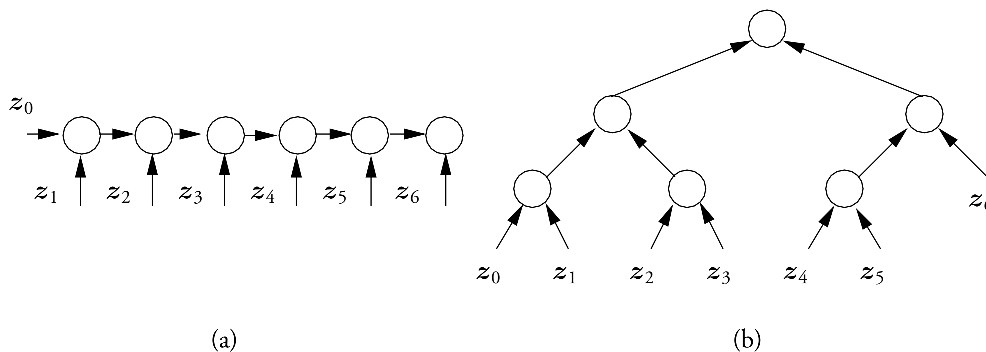


Figure 2.16 Two methods for aggregating the binary numbers $\boldsymbol{z_0}, \dots, \boldsymbol{z_{n-1}}$.

2.9.1 Carry-Save Multiplication

We now describe a much faster circuit obtained through the use of the carry-save adder. Let $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$ be three binary n-bit numbers. Their sum is a binary number $\boldsymbol{t}$. It follows that $|\boldsymbol{t}|$ can be represented as

$$\begin{aligned}|\boldsymbol{t}| &= |\boldsymbol{u}| + |\boldsymbol{v}| + |\boldsymbol{w}| \\ &= \sum_{i=0}^{n-1}(u_i + v_i + w_i)2^i\end{aligned}$$

With a full adder the sum $(u_i + v_i + w_i)$ can be converted to the binary representation $c_{i+1}2 + s_i$. Making this substitution, we have the following expression for the sum:

$$\begin{aligned}|\boldsymbol{t}| &= |\boldsymbol{u}| + |\boldsymbol{v}| + |\boldsymbol{w}| \\ &= \sum_{i=0}^{n-1}(2c_{i+1} + s_i)2^i \\ &= |\boldsymbol{c}| + |\boldsymbol{s}|\end{aligned}$$

Here $\boldsymbol{c}$ with $c_0 = 0$ is an $(n+1)$-tuple and $\boldsymbol{s}$ is an n-tuple. The conversion of (u_i, v_i, w_i) to (c_{i+1}, s_i) can be done with the full adder circuit shown in Fig. 2.15 of size 5 and depth 3 over the basis $\Omega = \{\wedge, \vee, \oplus\}$.

The function $f^{(n)}_{\text{carry-save}} : \mathcal{B}^{3n} \mapsto \mathcal{B}^{2n+2}$ that maps three binary n-tuples, $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$, to the pair $(\boldsymbol{c}, \boldsymbol{s})$ described above is the **carry-save adder**. A circuit of full adders that realizes this function is shown in Fig. 2.17.

THEOREM 2.9.1 *The carry-save adder function* $f^{(n)}_{\text{carry-save}} : \mathcal{B}^{3n} \mapsto \mathcal{B}^{2n+2}$ *can be realized with the following size and depth over the basis* $\Omega = \{\wedge, \vee, \oplus\}$*:*

$$\begin{aligned}C_\Omega\left(f^{(n)}_{\text{carry-save}}\right) &\leq 5n \\ D_\Omega\left(f^{(n)}_{\text{carry-save}}\right) &\leq 3\end{aligned}$$

Three binary n-bit numbers $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$ can be added by combining them in a carry-save adder to produce the pair $(\boldsymbol{c}, \boldsymbol{s})$, which are then added in an $(n+1)$-input binary adder. Any adder can be used for this purpose.

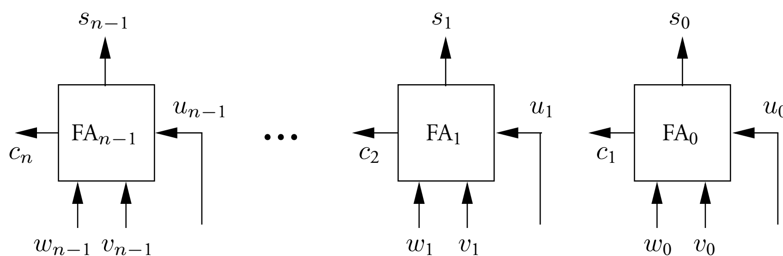


Figure 2.17 A carry-save adder realized by an array of full adders.

A multiplier for two n-bit binary can be formed by first creating the n $(2n-1)$-bit binary numbers shown in (2.8) and then adding them, as explained above. These n numbers can be added in groups of three, as suggested in Fig. 2.18.

Let's now count the number of levels of carry-save adders in this construction. At the zeroth level there are $m_0 = n$ numbers. At the jth level there are

$$m_j = 2\lfloor m_{j-1}/3 \rfloor + m_{j-1} - 3\lfloor m_{j-1}/3 \rfloor = m_{j-1} - \lfloor m_{j-1}/3 \rfloor$$

binary numbers. This follows because there are $\lfloor m_{j-1}/3 \rfloor$ groups of three binary numbers and each group is mapped to two binary numbers. Not combined into such groups are $m_{j-1} - \lfloor m_{j-1}/3 \rfloor$ binary numbers, giving the total m_j. Since $(x-2)/3 \le \lfloor x/3 \rfloor \le x/3$, we have

$$\left(\frac{2}{3}\right) m_{j-1} \le m_j \le \left(\frac{2}{3}\right) m_{j-1} + \left(\frac{2}{3}\right)$$

from which it is easy to show by induction that the following inequality holds:

$$\left(\frac{2}{3}\right)^j n \le m_j \le \left(\frac{2}{3}\right)^j n + 2\left(1 - \left(\frac{2}{3}\right)^j\right) \le \left(\frac{2}{3}\right)^j n + 2$$

Let s be the number of stages after which $m_s = 2$. Since $m_{s-1} \ge 3$, we have

$$\frac{\log_2(n/2)}{\log_2(3/2)} \le s \le \frac{\log_2 n}{\log_2(3/2)} + 1$$

The number of carry-save adders used in this construction is $n-2$. This follows from the observation that the number of carry-save adders used in one stage is equal to the decrease in the number of binary numbers from one stage to the next. Since we start with n and finish with 2, the result follows.

After reducing the n binary numbers to two binary numbers through a series of carry-save adder stages, the two remaining binary numbers are added in a traditional binary adder. Since each carry-save adder operates on three $(2n-1)$-bit binary numbers, they use at most $5(2n-1)$ gates and have depth 3. Summarizing, we have the following theorem showing that carry-save addition provides a multiplication circuit of depth $O(\log n)$ but of size quadratic in n.

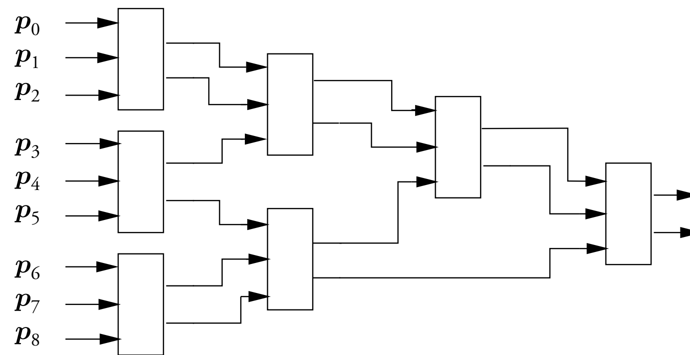


Figure 2.18 Schema for the carry-save combination of nine 18-bit numbers.

THEOREM 2.9.2 *The binary multiplication function* $f^{(n)}_{\mathrm{mult}} : \mathcal{B}^{2n} \mapsto \mathcal{B}^{2n}$ *for* n*-bit binary numbers can be realized by carry-save addition by a circuit of the following size and depth over the basis* $\Omega = \{\wedge, \vee, \oplus\}$*:*

$$C_\Omega\left(f^{(n)}_{\mathrm{mult}}\right) \leq 5(2n-1)(n-2) + C_\Omega\left(f^{(2n)}_{\mathrm{add}}\right)$$
$$D_\Omega\left(f^{(n)}_{\mathrm{mult}}\right) \leq 3s + D_\Omega\left(f^{(2n)}_{\mathrm{add}}\right)$$

where s*, the number of carry-save adder stages, satisfies*

$$s \leq \frac{\log_2 n}{\log_2(3/2)} + 1$$

It follows from this theorem and the results of Theorem 2.7.2 that two n-bit binary numbers can be multiplied by a circuit of size $O(n^2)$ and depth $O(\log n)$.

2.9.2 Divide-and-Conquer Multiplication

We now examine a multiplier of much smaller circuit size but depth $O(\log^2 n)$. It uses a **divide-and-conquer** technique. We represent two positive integers by their n-bit binary numbers $\boldsymbol{u}$ and $\boldsymbol{v}$. We assume that n is even and decompose each number into two $(n/2)$-bit numbers:

$$\boldsymbol{u} = (\boldsymbol{u}_h, \boldsymbol{u}_l), \quad \boldsymbol{v} = (\boldsymbol{v}_h, \boldsymbol{v}_l)$$

where $\boldsymbol{u}_h$, $\boldsymbol{u}_l$, $\boldsymbol{v}_h$, $\boldsymbol{v}_l$ are the high and low components of the vectors $\boldsymbol{u}$ and $\boldsymbol{v}$, respectively. Then we can write

$$|\boldsymbol{u}| = |\boldsymbol{u}_h| 2^{n/2} + |\boldsymbol{u}_l|$$
$$|\boldsymbol{v}| = |\boldsymbol{v}_h| 2^{n/2} + |\boldsymbol{v}_l|$$

from which we have

$$|\boldsymbol{u}||\boldsymbol{v}| = |\boldsymbol{u}_l||\boldsymbol{v}_l| + (|\boldsymbol{u}_h||\boldsymbol{v}_h| + (|\boldsymbol{v}_h| - |\,\boldsymbol{v}_l|)(|\boldsymbol{u}_l| - |\,\boldsymbol{u}_h|) + |\boldsymbol{u}_l||\boldsymbol{v}_l|)2^{n/2} + |\boldsymbol{u}_h||\boldsymbol{v}_h|2^n$$

It follows from this expression that only three integer multiplications are needed, namely $|\boldsymbol{u}_l||\boldsymbol{u}_l|$, $|\boldsymbol{u}_h||\boldsymbol{u}_h|$, and $(|\boldsymbol{v}_h| - |\,\boldsymbol{v}_l|)(|\boldsymbol{u}_l| - |\,\boldsymbol{u}_h|)$; multiplication by a power of 2 is done by realigning bits for addition. Each multiplication is of $(n/2)$-bit numbers. Six additions and subtractions of $2n$-bit numbers suffice to complete the computation. Each of the additions and subtractions can be done with a linear number of gates in logarithmic time.

If $C(n)$ and $D(n)$ are the size and depth of a circuit for integer multiplication realized with this divide-and-conquer method, then we have

$$C(n) \leq 3C(n/2) + cn \tag{2.9}$$
$$D(n) \leq D(n/2) + d\log_2 n \tag{2.10}$$

where c and d are constants of the construction. Since $C(1) = 1$ and $D(1) = 1$ (one use of AND suffices), we have the following theorem, the proof of which is left as an exercise (see Problem 2.28).

THEOREM 2.9.3 *If $n = 2^k$, the binary multiplication function $f^{(n)}_{\text{mult}} : \mathcal{B}^{2n} \mapsto \mathcal{B}^{2n}$ for n-bit binary numbers can be realized by a circuit for the divide-and-conquer algorithm of the following size and depth over the basis $\Omega = \{\wedge, \vee, \oplus\}$:*

$$C_\Omega\left(f^{(n)}_{\text{mult}}\right) = O\left(3^{\log_2 n}\right) = O\left(n^{\log_2 3}\right)$$
$$D_\Omega\left(f^{(n)}_{\text{mult}}\right) = O(\log_2^2 n)$$

The size of this divide-and-conquer multiplication circuit is $O(n^{1.585})$, which is much smaller than the $O(n^2)$ bound based on carry-save addition. The depth bound can be reduced to $O(\log n)$ through the use of carry-save addition. (See Problem 2.29.) However, even faster multiplication algorithms are known for large n.

2.9.3 Fast Multiplication

Schönhage and Strassen [302] have described a circuit to multiply integers represented in binary that is asymptotically small and shallow. Their algorithm for the multiplication of n-bit binary numbers uses $O(n \log n \log\log n)$ gates and depth $O(\log n)$. It illustrates the point that a circuit can be devised for this problem that has depth $O(\log n)$ and uses a number of gates considerably less than quadratic in n. Although the coefficients on the size and depth bounds are so large that their circuit is not practical, their result is interesting and motivates the following definition.

DEFINITION 2.9.1 *$M_{\text{int}}(n, c)$ is the size of the smallest circuit for the multiplication of two n-bit binary numbers that has depth at most $c \log_2 n$ for $c > 0$.*

The Schönhage-Strassen circuit demonstrates that $M_{\text{int}}(n, c) = O(n \log n \log\log n)$ for all $n \geq 1$. It is also clear that $M_{\text{int}}(n, c) = \Omega(n)$ because any multiplication circuit must examine each component of each binary number and no more than a constant number of inputs can be combined by one gate. (Chapter 9 provides methods for deriving lower bounds on the size and depth of circuits.)

Because we use integer multiplication in other circuits, it is convenient to make the following reasonable assumption about the dependence of $M_{\text{int}}(n, c)$ on n. We assume that

$$M_{\text{int}}(dn, c) \leq d M_{\text{int}}(n, c)$$

for all d satisfying $0 \leq d \leq 1$. This condition is satisfied by the Schönhage-Strassen circuit.

2.9.4 Very Fast Multiplication

If integers in the set $\{0, 1, \dots, N - 1\}$ are represented by the exponents of primes in their prime factorization, they can be multiplied by adding exponents. The largest exponent on a prime in this range is at most $\log_2 N$. Thus, exponents can be represented by $O(\log\log N)$ bits and integers multiplied by circuits with depth $O(\log\log\log N)$. (See Problem 2.32.) This depth is much smaller than $O(\log\log N)$, the depth of circuits to add integers in any fixed radix system. (Note that if $N = 2^n$, $\log_2 \log_2 N = \log_2 n$.) However, addition is very difficult in this number system. Thus, it is a fast number system only if the operations are limited to multiplications.

2.9.5 Reductions to Multiplication

The logical shifting function $f_{\mathrm{shift}}^{(n)}$ can be reduced to integer multiplication function $f_{\mathrm{mult}}^{(n)}$, as can be seen by letting one of the two n-tuple arguments be a power of 2. That is,

$$f_{\mathrm{shift}}^{(n)}(\boldsymbol{x}, \boldsymbol{s}) = \pi_L^{(n)}\left(f_{\mathrm{mult}}^{(n)}(\boldsymbol{x}, \boldsymbol{y})\right)$$

where $\boldsymbol{y} = f_{\mathrm{decode}}^{(m)}(\boldsymbol{s})$ is the value of the decoder function (see Section 2.5) that maps a binary m-tuple, $m = \lceil \log_2 n \rceil$, into a binary 2^m-tuple containing a single 1 at the output indexed by the integer represented by $\boldsymbol{s}$ and $\pi_L^{(n)}$ is the projection operator defined on page 50.

LEMMA 2.9.1 *The logical shifting function $f_{\mathrm{shift}}^{(n)}$ can be reduced to the binary integer multiplication function $f_{\mathrm{mult}}^{(n)}$ through the application of the decoder function $f_{\mathrm{decode}}^{(m)}$ on $m = \lceil \log_2 n \rceil$ inputs.*

As shown in Section 2.5, the decoder function $f_{\mathrm{decode}}^{(m)}$ can be realized with a circuit of size very close to 2^m and depth $\lceil \log_2 m \rceil$. Thus, the shifting function has circuit size and depth no more than constant factors larger than those for integer multiplication.

The **squaring function** $f_{\mathrm{square}}^{(n)} : \mathcal{B}^n \mapsto \mathcal{B}^{2n}$ maps the binary n-tuple $\boldsymbol{x}$ into the binary $2n$-tuple $\boldsymbol{y}$ representing the product of $\boldsymbol{x}$ with itself. Since the squaring and integer multiplication functions contain each other as subfunctions, as shown below, circuits for one can be used for the other.

LEMMA 2.9.2 *The integer multiplication function $f_{\mathrm{mult}}^{(n)}$ contains the squaring function $f_{\mathrm{square}}^{(n)}$ as a subfunction and $f_{\mathrm{square}}^{(3n+1)}$ contains $f_{\mathrm{mult}}^{(n)}$ as a subfunction.*

Proof The first statement follows by setting the two n-tuple inputs of $f_{\mathrm{mult}}^{(n)}$ to be the input to $f_{\mathrm{square}}^{(n)}$. The second statement follows by examining the value of $f_{\mathrm{square}}^{(3n+1)}$ on the $(3n+1)$-tuple input $(\boldsymbol{xzy})$, where $\boldsymbol{x}$ and $\boldsymbol{y}$ are binary n-tuples and $\boldsymbol{z}$ is the zero binary $(n+1)$-tuple. Thus, $(\boldsymbol{xzy})$ denotes the value $a = 2^{2n+1}|\boldsymbol{x}| + |\boldsymbol{y}|$ whose square b is

$$b = 2^{4n+2}|\boldsymbol{x}|^2 + 2^{2n+2}|\boldsymbol{x}||\boldsymbol{y}| + |\boldsymbol{y}|^2$$

The value of the product $|\boldsymbol{x}||\boldsymbol{y}|$ can be read from the output because there is no carry into $2^{2n+2}|\boldsymbol{x}||\boldsymbol{y}|$ from $|\boldsymbol{y}|^2$, nor is there a carry into $2^{4n+2}|\boldsymbol{x}|^2$ from $2^{2n+2}|\boldsymbol{x}||\boldsymbol{y}|$, since $|\boldsymbol{x}|, |\boldsymbol{y}| \leq 2^n - 1$. ■

2.10 Reciprocal and Division

In this section we examine methods to divide integers represented in binary. Since the division of one integer by another generally cannot be represented with a finite number of bits (consider, for example, the value of 2/3), we must be prepared to truncate the result of a division. The division method presented in this section is based on Newton's method for finding a zero of a function.

Let $\boldsymbol{u} = (u_{n-1}, \ldots, u_1, u_0)$ and $\boldsymbol{v} = (v_{n-1}, \ldots, v_1, v_0)$ denote integers whose magnitudes are u and v. Then the division of one integer u by another v, u/v, can be obtained as the

result of taking the product of u with the reciprocal $1/v$. (See Problem 2.33.) For this reason, we examine only the computation of reciprocals of n-bit binary numbers. For simplicity we assume that n is a power of 2.

The reciprocal of the n-bit binary number $\boldsymbol{u} = (u_{n-1}, \ldots, u_1, u_0)$ representing the integer u is a fractional number r represented by the (possibly infinite) binary number $\boldsymbol{r} = (r_{-1}, r_{-2}, r_{-3}, \ldots)$, where

$$|\boldsymbol{r}| = r_{-1}2^{-1} + r_{-2}2^{-2} + r_{-3}2^{-3} + \cdots$$

Some numbers, such as 3, have a binary reciprocal that has an infinite number of digits, such as $(0, 1, 0, 1, 0, 1, \ldots)$, and cannot be expressed exactly as a binary tuple of finite extent. Others, such as 4, have reciprocals that have finite extent, such as $(0, 1)$.

Our goal is to produce an $(n + 2)$-bit approximation to the reciprocal of n-bit binary numbers. (It simplifies the analysis to obtain an $(n+2)$-bit approximation instead of an n-bit approximation.) We assume that each such binary number $\boldsymbol{u}$ has a 1 in its most significant position; that is, $2^{n-1} \leq u < 2^n$. If this is not true, a simple circuit can be devised to determine the number of places by which to shift $\boldsymbol{u}$ left to meet this condition. (See Problem 2.25.) The result is shifted left by an equal amount to produce the reciprocal.

It follows that an $(n+2)$-bit approximation to the reciprocal of an n-bit binary number $\boldsymbol{u}$ with $u_{n-1} = 1$ is represented by $\boldsymbol{r} = (r_{-1}, r_{-2}, r_{-3}, \ldots)$, where the first $n - 2$ digits of $\boldsymbol{r}$ are zero. Thus, the value of the approximate reciprocal is represented by the $n + 2$ components $(r_{-(n-1)}, r_{-(n)}, \ldots, r_{-(2n)})$. It follows that these components are produced by shifting $\boldsymbol{r}$ left by $2n$ places and removing the fractional bits. This defines the function $f^{(n)}_{\text{recip}}$:

$$f^{(n)}_{\text{recip}}(\boldsymbol{u}) = \left\lfloor \frac{2^{2n}}{u} \right\rfloor$$

The approximation described below can be used to compute reciprocals.

Newton's approximation algorithm is a method to find the zero x_0 of a twice continuously differentiable function $h : \mathbb{R} \mapsto \mathbb{R}$ on the reals (that is, $h(x_0) = 0$) when h has a non-zero derivative $h'(x)$ in the neighborhood of x_0. As suggested in Fig. 2.19, the slope of the tangent to the curve at the point y_i, $h'(y_i)$, is equal to $h(y_i)/(y_i - y_{i+1})$. For the convex increasing function shown in this figure, the value of y_{i+1} is closer to the zero x_0 than

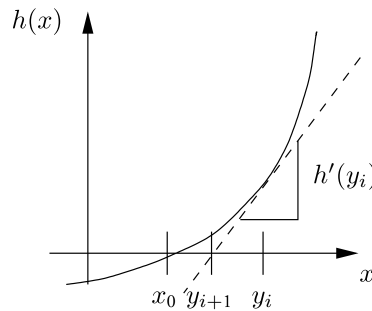


Figure 2.19 Newton's method for finding the zero of a function.

is y_i. The same holds for all twice continuously differentiable functions whether increasing, decreasing, convex, or concave in the neighborhood of a zero. It follows that the recurrence

$$y_{i+1} = y_i - \frac{h(y_i)}{h'(y_i)} \tag{2.11}$$

provides values increasingly close to the zero of h as long as it is started with a value sufficiently close to the zero.

The function $h(y) = 1 - 2^{2n}/uy$ has zero $y = 2^{2n}/u$. Since $h'(y) = 2^{2n}/uy^2$, the recurrence (2.11) becomes

$$y_{i+1} = 2y_i - uy_i^2/2^{2n}$$

When this recurrence is modified as follows, it converges to the $(n+2)$-bit binary reciprocal of the n-bit binary number $\boldsymbol{u}$:

$$y_{i+1} = \left\lfloor \frac{2^{2n+1}y_i - uy_i^2}{2^{2n}} \right\rfloor$$

The size and depth of a circuit resulting from this recurrence are $O(M_{\text{int}}(n, c)\log n)$ and $O(\log^2 n)$, respectively. However, this recurrence uses more gates than are necessary since it does calculations with full precision at each step even though the early steps use values of y_i that are imprecise. We can reduce the size of the resulting circuit to $O(M_{\text{int}}(n, c))$ if, instead of computing the reciprocal with $n + 2$ bits of accuracy at every step we let the amount of accuracy vary with the number of stages, as in the algorithm $\texttt{recip}(u, n)$ of Fig. 2.20. The algorithm $\texttt{recip}$ is called $1 + \log_2 n$ times, the last time when $n = 1$.

We now show that the algorithm $\texttt{recip}(u, n)$ computes the function $f^{(n)}_{\text{recip}}(\boldsymbol{u}) = r = \lfloor 2^{2n}/u \rfloor$. In other words, we show that r satisfies $ru = 2^{2n} - s$ for some $0 \le s < u$. The proof is by induction on n.

The inductive hypothesis is that the algorithm $\texttt{recip}(u, m)$ produces an $(m + 2)$-bit approximation to the reciprocal of the m-bit binary number u (whose most significant bit is 1), that is, it computes $r = \lfloor 2^{2m}/u \rfloor$. The assumption applies to the base case of $m = 1$ since $u = 1$ and $r = 4$. We assume it holds for $m = n/2$ and show that it also holds for $m = n$.

```
Algorithm recip(u, n)
    if n = 1 then
        r := 4;
    else begin
        t := recip(⌊u/2^{n/2}⌋, n/2);
        r := ⌊(2^{3n/2 + 1} t − u t^2)/2^n⌋ ;
        for j := 3 downto 0 do
            if (u(r + 2^j) ≤ 2^{2n}) then r := r + 2^j ;
    end;
return(r);
```

Figure 2.20 An algorithm to compute **r**, the $(n+2)$-bit approximation to the reciprocal of the n-bit binary number **u** representing the integer u, that is, $\mathbf{r} = f^{(n)}_{\text{recip}}(\mathbf{u})$.

Let u_1 and u_0 be the integers corresponding to the most and least significant $n/2$ bits respectively of u, that is, $u = u_1 2^{n/2} + u_0$. Since $2^{n-1} \leq u < 2^n$, $2^{n/2-1} \leq u_1 < 2^{n/2}$. Also, $\lfloor \frac{u}{2^{n/2}} \rfloor = u_1$. By the inductive hypothesis $t = \lfloor 2^n/u_1 \rfloor$ is the value returned by $\texttt{recip}(u_1, n/2)$; that is, $u_1 t = 2^n - s'$ for some $0 \leq s' < u_1$. Let $w = 2^{3n/2\,+\,1}t - ut^2$. Then

$$uw = 2^{2n+1}u_1 t + 2^{3n/2\,+\,1}u_0 t - [t(u_1 2^{n/2} + u_0)]^2$$

Applying $u_1 t = 2^n - s'$, dividing both sides by 2^n, and simplifying yields

$$\frac{uw}{2^n} = 2^{2n} - \left(s' - \frac{tu_0}{2^{n/2}}\right)^2 \tag{2.12}$$

We now show that

$$\frac{uw}{2^n} \geq 2^{2n} - 8u \tag{2.13}$$

by demonstrating that $(s' - tu_0/2^{n/2})^2 \leq 8u$. We note that $s' < u_1 < 2^{n/2}$, which implies $(s')^2 < 2^{n/2}u_1 \leq u$. Also, since $u_1 t = 2^n - s'$ or $t \leq 2^n/u_1$ we have

$$\left(\frac{tu_0}{2^{n/2}}\right)^2 < \left(\frac{2^{n/2}u_0}{u_1}\right)^2 < \left(2^{n/2+1}\right)^2 \leq 8u$$

since $u_1 \geq 2^{n/2-1}$, $u_0 < 2^{n/2}$, and $2^{n-1} \leq u$. The desired result follows from the observation that $(a-b)^2 \leq \max(a^2, b^2)$.

Since $r = \lfloor w/2^n \rfloor$, it follows from (2.13) that

$$ur = u\left\lfloor \frac{w}{2^n} \right\rfloor > u\left(\frac{w}{2^n} - 1\right) = \frac{uw}{2^n} - u \geq 2^{2n} - 9u$$

It follows that $r > (2^{2n}/u) - 9$. Also from (2.12), we see that $r \leq 2^{2n}/u$. The three-step adjustment process at the end of $\texttt{recip}(u, m)$ increases ur by the largest integer multiple of u less than $16u$ that keeps it less than or equal to 2^{2n}. That is, r satisfies $ur = 2^{2n} - s$ for some $0 \leq s < u$, which means that r is the reciprocal of u.

The algorithm for $\texttt{recip}(u, n)$ translates into a circuit as follows: a) $\texttt{recip}(u, 1)$ is realized by an assignment, and b) $\texttt{recip}(u, n)$, $n > 1$, is realized by invoking a circuit for $\texttt{recip}(\lfloor \frac{u}{2^{n/2}} \rfloor, n/2)$ followed by a circuit for $\lfloor (2^{3n/2\,+\,1}t - ut^2)/2^n \rfloor$ and one to implement the three-step adjustment. The first of these steps computes $\lfloor \frac{u}{2^{n/2}} \rfloor$, which does not require any gates, merely shifting and discarding bits. The second step requires shifting t left by $3n/2$ places, computing t^2 and multiplying it by u, subtracting the result from the shifted version of t, and shifting the final result right by n places and discarding low-order bits. Circuits for this have size $cM_{\text{int}}(n, c)$ for some constant $c > 0$ and depth $O(\log n)$. The third step can be done by computing ur, adding $u2^j$ for $j = 3, 2, 1$, or 0, and comparing the result with 2^{2n}. The comparisons control whether 2^j is added to r or not. The one multiplication and the additions can be done with circuits of size $c'M_{\text{int}}(n, c)$ for some constant $c' > 0$ and depth $O(\log n)$. The comparison operations can be done with a constant additional number of gates and constant depth. (See Problem 2.19.)

It follows that $\texttt{recip}$ can be realized by a circuit whose size $C_{\text{recip}}(n)$ is no more than a multiple of the size of an integer multiplication circuit, $M_{\text{int}}(n, c)$, plus the size of a circuit for

the invocation of `recip(`$\lfloor \frac{u}{2^{n/2}} \rfloor$`,n/2)`. That is,

$$C_{\text{recip}}(n) \leq C_{\text{recip}}(n/2) + cM_{\text{int}}(n,c)$$
$$C_{\text{recip}}(1) = 1$$

for some constant $c > 0$. This inequality implies the following bound:

$$C_{\text{recip}}(n) \leq c \sum_{j=0}^{\log n} M_{\text{int}}\left(\frac{n}{2^j}, c\right) \leq cM_{\text{int}}(n,c) \sum_{j=0}^{\log n} \frac{1}{2^j}$$
$$= O(M_{\text{int}}(n,c))$$

which follows since $M_{\text{int}}(dn,c) \leq dM_{\text{int}}(n,c)$ when $d \leq 1$.

The depth $D_{\text{recip}}(n)$ of the circuit produced by this algorithm is at most $c \log n$ plus the depth $D_{\text{recip}}(n/2)$. Since the circuit has at most $1 + \log_2 n$ stages with a depth of at most $c \log n$ each, $D_{\text{recip}}(n) \leq 2c \log^2 n$ when $n \geq 2$.

THEOREM 2.10.1 *If $n = 2^k$, the reciprocal function $f^{(n)}_{\text{recip}} : \mathcal{B}^n \mapsto \mathcal{B}^{n+2}$ for n-bit binary numbers can be realized by a circuit with the following size and depth:*

$$C_\Omega\left(f^{(n)}_{\text{recip}}\right) \leq O(M_{\text{int}}(n,c))$$
$$D_\Omega\left(f^{(n)}_{\text{recip}}\right) \leq c \log_2^2 n$$

VERY FAST RECIPROCAL Beame, Cook, and Hoover [33] have given an $O(\log n)$ circuit for the reciprocal function. It uses a sequence of about $n^2/\log n$ primes to represent an n-bit binary number $\boldsymbol{x}$, $.5 \leq \boldsymbol{x} < 1$, using arithmetic modulo these primes. The size of the circuit produced is polynomial in n, although much larger than $M_{\text{int}}(n,c)$. Reif and Tate [324] show that the reciprocal function can be computed with a circuit that is defined only in terms of n and has a size proportional to M_{int} (and thus nearly optimal) and depth $O(\log n \log \log n)$. Although the depth bound is not quite as good as that of Beame, Cook, and Hoover, its size bound is very good.

2.10.1 Reductions to the Reciprocal

In this section we show that the reciprocal function contains the squaring function as a subfunction. It follows from Problem 2.33 and the preceding result that integer multiplication and division have comparable circuit size. We use Taylor's theorem [314, p. 345] to establish the desired result.

THEOREM 2.10.2 (Taylor) *Let $f(x) : \mathbb{R} \mapsto \mathbb{R}$ be a continuous real-valued function defined on the interval $[a,b]$ whose kth derivative is also continuous for $k \leq n+1$ over the same interval. Then for $a \leq x_0 \leq x \leq b$, $f(x)$ can be expanded as*

$$f(x) = f(x_0) + (x - x_0)f^{[1]}(x_0) + \frac{(x-x_0)^2}{2} f^{[2]}(x_0) + \cdots + \frac{(x-x_0)^n}{n!} f^{[n]}(x_0) + r_n$$

where $f^{[n]}$ denotes the nth derivative of f and the remainder r_n satisfies

$$r_n = \int_{x_0}^{x} f^{[n+1]}(t) \frac{(x-t)^n}{n!} \, dt$$

$$= \frac{(x-x_0)^{n+1}}{(n+1)!} f^{[n+1]}(\psi)$$

for some ψ satisfying $x_0 \leq \psi \leq x$.

Taylor's theorem is used to expand $\lfloor 2^{2n-1}/|\boldsymbol{u}| \rfloor$ by applying it to the function $f(w) = (1+w)^{-1}$ on the interval $[0,1]$. The Taylor expansion of this function is

$$(1+w)^{-1} = 1 - w + w^2 - w^3(1+\psi)^{-4}$$

for some $0 \leq \psi \leq 1$. The magnitude of the last term is at most w^3.

Let $n \geq 12$, $k = \lfloor n/2 \rfloor$, $l = \lfloor n/12 \rfloor$ and restrict $|\boldsymbol{u}|$ as follows:

$$\begin{aligned} |\boldsymbol{u}| &= 2^k + |\boldsymbol{a}| \text{ where} \\ |\boldsymbol{a}| &= 2^l|\boldsymbol{b}| + 1 \text{ and} \\ |\boldsymbol{b}| &\leq 2^{l-1} - 1 \end{aligned}$$

It follows that $|\boldsymbol{a}| \leq 2^{2l-1} - 2^l + 1 < 2^{2l-1}$ for $l \geq 1$. Applying the Taylor series expansion to $(1 + |\boldsymbol{a}|/2^k)^{-1}$, we have

$$\left\lfloor \frac{2^{2n-1}}{(2^k + |\boldsymbol{a}|)} \right\rfloor = \left\lfloor 2^{2n-1-k} \left(1 - \frac{|\boldsymbol{a}|}{2^k} + \left(\frac{|\boldsymbol{a}|}{2^k}\right)^2 - \left(\frac{|\boldsymbol{a}|}{2^k}\right)^3 (1+\psi)^{-4} \right) \right\rfloor \quad (2.14)$$

for some $0 \leq \psi \leq 1$. For the given range of values for $|\boldsymbol{u}|$ both the sum of the first two terms and the third term on the right-hand side have the following bounds:

$$\begin{aligned} 2^{2n-1-k}(1 - |\boldsymbol{a}|/2^k) &> 2^{2n-1-k}\left(1 - 2^{2l-1}/2^k\right) \\ 2^{2n-1-k}(|\boldsymbol{a}|/2^k)^2 &< \ 2^{2n-1-k}\left(2^{2l-1}/2^k\right)^2 \end{aligned}$$

Since $2^{2l-1}/2^k < 1/2$, the value of the third term, $2^{2n-1-k}(|\boldsymbol{a}|/2^k)^2$, is an integer that does not overlap in any bit positions with the sum of the first two terms.

The fourth term is negative; its magnitude has the following upper bound:

$$2^{2n-1-4k}|\boldsymbol{a}|^3(1+\psi)^{-4} < 2^{3(2l-1)+2n-1-4k}$$

Expanding the third term, we have

$$2^{2n-1-3k}(|\boldsymbol{a}|)^2 = 2^{2n-1-3k}(2^{2l}|\boldsymbol{b}|^2 + 2^{l+1}|\boldsymbol{b}| + 1)$$

Because $3(2l-1) \leq k$, the third term on the right-hand side of this expansion has value $2^{2n-1-3k}$ and is larger than the magnitude of the fourth term in (2.14). Consequently the fourth term does not affect the value of the result in (2.14) in positions occupied by the binary representation of $2^{2n-1-3k}(2^{2l}|\boldsymbol{b}|^2 + 2^{l+1}|\boldsymbol{b}|)$. In turn, $2^{l+1}|\boldsymbol{b}|$ is less than 2^{2l}, which means that the binary representation of $2^{2n-1-3k}(2^{2l}|\boldsymbol{b}|^2)$ appears in the output shifted but otherwise without modification. This provides the following result.

LEMMA 2.10.1 *The reciprocal function $f^{(n)}_{\text{recip}}$ contains as a subfunction the squaring function $f^{(m)}_{\text{square}}$ for $m = \lfloor n/12 \rfloor - 1$.*

Proof The value of the l-bit binary number denoted by $\boldsymbol{b}$ appears in the output if $l = \lfloor n/12 \rfloor \geq 1$. ■

Lower bounds similar to those derived for the reciprocal function can be derived for special fractional powers of binary numbers. (See Problem 2.35.)

2.11 Symmetric Functions

The symmetric functions are encountered in many applications. Among the important symmetric functions is binary sorting, the binary version of the standard sorting function. A surprising fact holds for binary sorting, namely, that it can be realized on n inputs by a circuit whose size is linear in n (see Problem 2.17), whereas non-binary sorting requires on the order of $n \log n$ operations. Binary sorting, and all other symmetric functions, can be realized efficiently through the use of a counting circuit that counts the number of 1's among the n inputs with a circuit of size linear in n. The counting circuit uses AND, OR, and NOT. When negations are disallowed, binary sorting requires on the order of $n \log n$ gates, as shown in Section 9.6.1.

DEFINITION 2.11.1 *A* **permutation** π *of an n-tuple* $\boldsymbol{x} = (x_1, x_2, \dots, x_n)$ *is a reordering* $\pi(\boldsymbol{x}) = (x_{\pi(1)}, x_{\pi(2)}, \dots, x_{\pi(n)})$ *of the components of* $\boldsymbol{x}$*. That is,* $\{\pi(1), \pi(2), \dots, \pi(n)\} = \{1, 2, 3, \dots, n\}$*. A* **symmetric function** $f^{(n)} : \mathcal{B}^n \mapsto \mathcal{B}^m$ *is a function for which* $f^{(n)}(\boldsymbol{x}) = f^{(n)}(\pi(\boldsymbol{x}))$ *for all permutations* π*.* $S_{n,m}$ *is the set of all symmetric functions* $f^{(n)} : \mathcal{B}^n \mapsto \mathcal{B}^m$ *and* $S_n = S_{n,1}$ *is the set of Boolean symmetric functions on n inputs.*

If $f^{(3)}$ is symmetric, then $f^{(3)}(0,1,1) = f^{(3)}(1,0,1) = f^{(3)}(1,1,0)$.

The following are symmetric functions:

1. **Threshold functions** $\tau_t^{(n)} : \mathcal{B}^n \mapsto \mathcal{B}$, $1 \le t \le n$:

$$\tau_t^{(n)}(\boldsymbol{x}) = \begin{cases} 1 & \sum_{j=1}^n x_j \ \ge \ t \\ 0 & \text{otherwise} \end{cases}$$

2. **Elementary symmetric functions** $e_t^{(n)} : \mathcal{B}^n \mapsto \mathcal{B}$, $0 \le t \le n$:

$$e_t^{(n)}(\boldsymbol{x}) = \begin{cases} 1 & \sum_{j=1}^n x_j \ = \ t \\ 0 & \text{otherwise} \end{cases}$$

3. **Binary sorting function** $f_{\text{sort}}^{(n)} : \mathcal{B}^n \mapsto \mathcal{B}^n$ sorts an n-tuple into descending order:

$$f_{\text{sort}}^{(n)}(\boldsymbol{x}) = (\tau_1^{(n)}, \tau_2^{(n)}, \dots, \tau_n^{(n)})$$

   Here $\tau_t^{(n)}$ is the tth threshold function.

4. **Modulus functions** $f_{c,\text{ mod } m}^{(n)} : \mathcal{B}^n \mapsto \mathcal{B}$, $0 \le c \le m-1$:

$$f_{c,\text{ mod } m}^{(n)}(\boldsymbol{x}) = \begin{cases} 1 & \sum_{j=1}^n x_j \ = \ c \bmod m \\ 0 & \text{otherwise} \end{cases}$$

The elementary symmetric functions e_t are building blocks in terms of which other symmetric functions can be realized at small additional cost. Each symmetric function $f^{(n)}$ is determined uniquely by its value v_t, $0 \le t \le n$, when exactly t of the input variables are 1. It follows that we can write $f^{(n)}(\boldsymbol{x})$ as

$$f^{(n)}(\boldsymbol{x}) = \bigvee_{0 \le t \le n} v_t \wedge e_t^{(n)}(\boldsymbol{x}) = \bigvee_{t \,|\, v_t = 1} e_t^{(n)}(\boldsymbol{x}) \tag{2.15}$$

Thus, efficient circuits for the elementary symmetric functions yield efficient circuits for general symmetric functions.

An efficient circuit for the elementary symmetric functions can be obtained from a circuit for counting the number of 1's among the variables $\boldsymbol{x}$. This **counting function** $f^{(n)}_{\text{count}} : \mathcal{B}^n \mapsto \mathcal{B}^{\lceil \log_2(n+1) \rceil}$ produces a $\lceil \log_2(n+1) \rceil$-bit binary number representing the number of 1's among the n inputs $x_1, x_2, \dots, x_n$.

A recursive construction for the counting function is shown in Fig. 2.21 (b) when $m = 2^{l+1} - 1$. The m inputs are organized into three groups, the first $2^l - 1$ Boolean variables $\boldsymbol{u}$, the second $2^l - 1$ variables $\boldsymbol{v}$, and the last variable x_m. The sum is represented by l "sum bits" $s^{(l+1)}_j$, $0 \leq j \leq l-1$, and the "carry bit" $c^{(l+1)}_{l-1}$. This sum is formed by adding in a ripple adder the outputs $s^{(l)}_j$, $0 \leq j \leq l-2$, and $c^{(l+1)}_l$ from the two counting circuits, each on $2^l - 1$ inputs, and the mth input x_m. (We abuse notation and use the same variables for the outputs of the different counting circuits.) The counting circuit on $2^2 - 1 = 3$ inputs is the full adder of Fig. 2.21(a). From this construction we have the following theorem:

LEMMA 2.11.1 *For $n = 2^k - 1$, $k \geq 2$, the counting function $f^{(n)}_{\text{count}} : \mathcal{B}^n \mapsto \mathcal{B}^{\lceil \log_2(n+1) \rceil}$ can be realized with the following circuit size and depth over the basis $\Omega = \{\wedge, \vee, \oplus\}$:*

$$C_\Omega\left(f^{(n)}_{\text{count}}\right) \leq 5(2^k - k - 1)$$
$$D_\Omega\left(f^{(n)}_{\text{count}}\right) \leq 4k - 5$$

Proof Let $C(k) = C_\Omega\left(f^{(n)}_{\text{count}}\right)$ and $D(k) = D_\Omega\left(f^{(n)}_{\text{count}}\right)$ when $n = 2^k - 1$. Clearly, $C(2) = 5$ and $D(2) = 3$ since a full adder over $\Omega = \{\wedge, \vee, \oplus\}$ has five gates and depth 3. The following inequality is immediate from the construction:

$$C(k) \leq 2C(k-1) + 5(k-1)$$

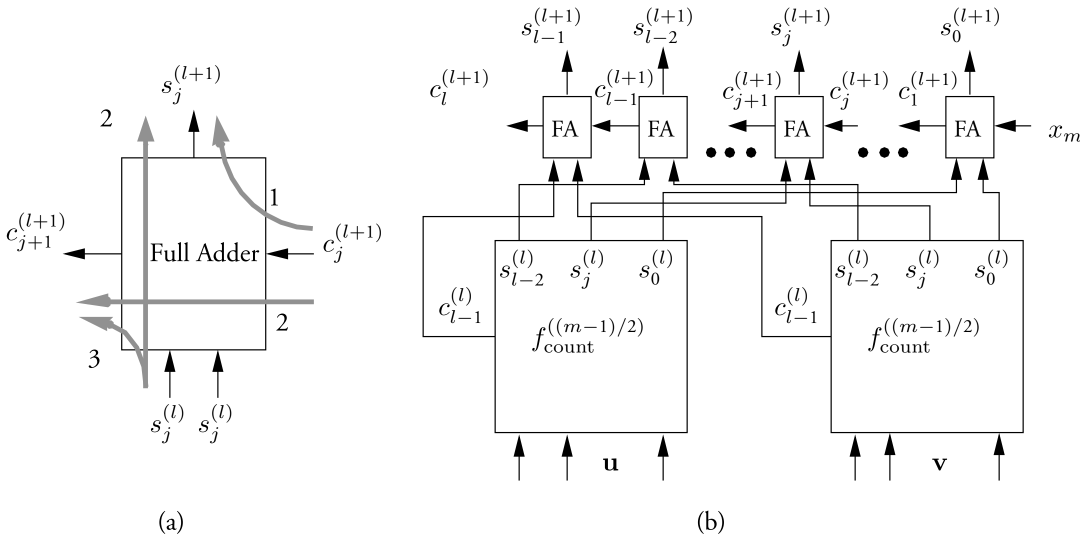


Figure 2.21 A recursive construction for the counting function $f^{(m)}_{\text{count}}$, $m = 2^{l+1} - 1$.

The size bound follows immediately. The depth bound requires a more careful analysis.

Shown in Fig. 2.21(a) is a full adder together with notation showing the amount by which the length of a path from one input to another is increased in passing through it when the full-adder circuit used is that shown in Fig. 2.14 and described by Equation 2.6. From this it follows that

$$D_\Omega\left(c_{j+1}^{(l+1)}\right) = \max\left(D_\Omega\left(c_j^{(l+1)}\right)+2, D_\Omega\left(s_j^{(l)}\right)+3\right)$$
$$D_\Omega\left(s_j^{(l+1)}\right) = \max\left(D_\Omega\left(c_j^{(l+1)}\right)+1, D_\Omega\left(s_j^{(l)}\right)+2\right)$$

for $2 \leq l$ and $0 \leq j \leq l-1$, where $s_{l-1}^{(l)} = c_{l-1}^{(l)}$. It can be shown by induction that $D_\Omega\left(c_j^{(k)}\right) = 2(k+j)-3$, $1 \leq j \leq k-1$, and $D_\Omega\left(s_j^{(k)}\right) = 2(k+j)-2$, $0 \leq j \leq k-2$, both for $2 \leq k$. (See Problem 2.16.) Thus, $D_\Omega\left(f_{\text{count}}^{(n)}\right) = D_\Omega\left(c_{k-1}^{(k)}\right) = (4k-5)$. ■

We now use this bound to derive upper bounds on the size and depth of symmetric functions in the class $S_{n,m}$.

THEOREM 2.11.1 *Every symmetric function $f^{(n)} : \mathcal{B}^n \mapsto \mathcal{B}^m$ can be realized with the following circuit size and depth over the basis $\Omega = \{\wedge, \vee, \oplus\}$ where $\phi(k) = 5(2^k - k - 1)$:*

$$C_\Omega\left(f^{(n)}\right) \leq m\lceil (n+1)/2\rceil + \phi(k) + 2(n+1) + (2\lceil \log_2(n+1)\rceil - 2)\sqrt{2(n+1)}$$
$$D_\Omega\left(f^{(n)}\right) \leq 5\lceil \log_2(n+1)\rceil + \lceil \log_2\lceil \log_2(n+1)\rceil\rceil - 4$$

for $k = \lceil \log_2(n+1)\rceil$ even.

Proof Lemma 2.11.1 establishes bounds on the size and depth of the function $f_{\text{count}}^{(n)}$ for $n = 2^k - 1$. For other values of n, let $k = \lceil \log_2(n+1)\rceil$ and fill out the $2^k - 1 - n$ variables with 0's.

The elementary symmetric functions are obtained by applying the value of $f_{\text{count}}^{(n)}$ as argument to the decoder function. A circuit for this function has been constructed that has size $2(n+1) + (2\lceil \log_2(n+1)\rceil - 2)\sqrt{2(n+1)}$ and depth $\lceil \log_2\lceil \log_2(n+1)\rceil\rceil + 1$. (See Lemma 2.5.4. We use the fact that $2^{\lceil \log_2 m\rceil} \leq 2m$.) Thus, all elementary symmetric functions on n variables can be realized with the following circuit size and depth:

$$C_\Omega\left(e_0^{(n)}, e_1^{(n)}, \ldots, e_n^{(n)}\right) \leq \phi(k) + 2(n+1) + (2\lceil \log_2(n+1)\rceil - 2)\sqrt{2(n+1)}$$
$$D_\Omega\left(e_0^{(n)}, e_1^{(n)}, \ldots, e_n^{(n)}\right) \leq 4k - 5 + \lceil \log_2\lceil \log_2(n+1)\rceil\rceil + 1$$

The expansion of Equation (2.15) can be used to realize an arbitrary Boolean symmetric function. Clearly, at most n OR gates and depth $\lceil \log_2 n\rceil$ suffice to realize each one of m arbitrary Boolean symmetric functions. (Since the v_t are fixed, no ANDs are needed.) This number of ORs can be reduced to $(n-1)/2$ as follows: if $\lceil (n+1)/2\rceil$ or more elementary functions are needed, use the complementary set (of at most $\lfloor (n+1)/2\rfloor$ functions) and take the complement of the result. Thus, no more than $\lceil (n+1)/2\rceil - 1$ ORs are needed per symmetric function (plus possibly one NOT), and depth at most $\lceil \log_2\lfloor ((n+1)/2)\rfloor\rceil + 1 \leq \lceil \log_2(n+1)\rceil$. ■

This theorem establishes that the binary sorting $f^{(n)}_{\mathrm{sort}} : \mathcal{B}^n \mapsto \mathcal{B}^n$ has size $O(n^2)$. In fact, a linear-size circuit can be constructed for it, as stated in Problem 2.17.

2.12 Most Boolean Functions Are Complex

As we show in this section, the circuit size and depth of most Boolean functions $f : \mathcal{B}^n \mapsto \mathcal{B}$ on n variables are at least exponential and linear in n, respectively. Furthermore, we show in Section 2.13 that such functions can be realized with circuits whose size and depth are at most exponential and linear, respectively, in n. Thus, the circuit size and depth of most Boolean functions on n variables are tightly bounded. Unfortunately, this result says nothing about the size and depth of a specific function, the case of most interest.

Each Boolean function on n variables is represented by a table with 2^n rows and one column of values for the function. Since each entry in this one column can be completed in one of two ways, there are 2^{2^n} ways to fill in the column. Thus, there are exactly 2^{2^n} Boolean functions on n variables. Most of these functions cannot be realized by small circuits because there just are not enough small circuits.

THEOREM 2.12.1 *Let $0 < \epsilon < 1$. The fraction of the Boolean functions $f : \mathcal{B}^n \mapsto \mathcal{B}$ that have size complexity $C_{\Omega_0}(f)$ satisfying the following lower bound is at least $1 - 2^{-(\epsilon/2)2^n}$ when $n \geq 2[(1-\epsilon)/\epsilon]\log_2[(3e)^2(1-\epsilon/2)]$. (Here $e = 2.71828\ldots$ is the base of the natural logarithm.)*

$$C_{\Omega_0}(f) \geq \frac{2^n}{n}(1-\epsilon) - 2n^2$$

Proof Each circuit contains some number, say g, of gates and each gate can be one of the three types of gate in the standard basis. The circuit with no gates computes the constant functions with value of 1 or 0 on all inputs.

An input to a gate can either be the output of another gate or one of the n input variables. (Since the basis Ω_0 is $\{\text{AND}, \text{OR}, \text{NOT}\}$, no gate need have a constant input.) Since each gate has at most two inputs, there are at most $(g-1+n)^2$ ways to connect inputs to one gate and $(g-1+n)^{2g}$ ways to interconnect g gates. In addition, since each gate can be one of three types, there are 3^g ways to name the gates. Since there are $g!$ orderings of g items (gates) and the ordering of gates does not change the function they compute, at most $N(g) = 3^g(g+n)^{2g}/g!$ distinct functions can be realized with g gates. Also, since $g! \geq g^g e^{-g}$ (see Problem 2.2) it follows that

$$N(g) \leq (3e)^g[(g^2+2gn+n^2)/g]^g \leq (3e)^g(g+2n^2)^g$$

The last inequality follows because $2gn + n^2 \leq 2gn^2$ for $n \geq 2$. Since the last bound is an increasing function of g, $N(0) = 2$ and $G + 1 \leq (3e)^G$ for $G \geq 1$, the number $M(G)$ of functions realizable with between 0 and G gates satisfies

$$M(G) \leq (G+1)(3e)^G(G+2n^2)^G \leq [(3e)^2(G+2n^2)]^G \leq (x^x)^{1/a}$$

where $x = a(G+2n^2)$ and $a = (3e)^2$. With base-2 logarithms, it is straightforward to show that $x^x \leq 2^{x_0}$ if $x \leq x_0/\log_2 x_0$ and $x_0 \geq 2$.

If $M(G) \leq 2^{(1-\delta)2^n}$ for $0 < \delta < 1$, at most a fraction $2^{(1-\delta)2^n}/2^{2^n} = 2^{-\delta 2^n}$ of the Boolean functions on n variables have circuits with G or fewer gates.

Let $G < 2^n(1-\epsilon)/n - 2n^2$. Then $x = a(G + 2n^2) \leq a2^n(1-\epsilon)/n \leq x_0/\log_2 x_0$ for $x_0 = a2^n(1-\epsilon/2)$ when $n \geq 2[(1-\epsilon)/\epsilon]\log_2[(3e)^2(1-\epsilon/2)]$, as can be shown directly. It follows that $M(G) \leq (x^x)^{1/a} \leq 2^{x_0} = 2^{2^n(1-\epsilon/2)}$. ■

To show that most Boolean functions $f : \mathcal{B}^n \mapsto \mathcal{B}$ over the basis Ω_0 require circuits with a depth linear in n, we use a similar argument. We first show that for every circuit there is a **tree circuit** (a circuit in which either zero or one edge is directed away from each gate) that computes the same function and has the same depth. Thus when searching for small-depth circuits it suffices to look only at tree circuits. We then obtain an upper bound on the number of tree circuits of depth d or less and show that unless d is linear in n, most Boolean functions on n variables cannot be realized with this depth.

LEMMA 2.12.1 *Given a circuit for a function $f : \mathcal{B}^n \mapsto \mathcal{B}^m$, a tree circuit can be constructed of the same depth that computes f.*

Proof Convert a circuit to a tree circuit without changing its depth as follows: find a vertex v with out-degree 2 or more at maximal distance from an output vertex. Attach a copy of the tree subcircuit with output vertex v to each of the edges directed away from v. This reduces by 1 the number of vertices with out-degree greater than 1 but doesn't change the depth or function computed. Repeat this process on the new circuit until no vertices of outdegree greater than 1 remain. ■

We count the number of tree circuits of depth d as follows. First, we determine $T(d)$, the number of binary, unlabeled, and unoriented trees of depth d. (The root has two descendants as does every other vertex except for leaves which have none. No vertex carries a label and we count as one tree those trees that differ only by the exchange of the two subtrees at a vertex.) We then multiply $T(d)$ by the number of ways to label the internal vertices with one of at most three gates and the leaves by at most one of n variables or constants to obtain an upper bound on $N(d)$, the number of distinct tree circuits of depth d. Since a tree of depth d has at most $2^d - 1$ internal vertices and 2^d leaves (see Problem 2.3), $N(d) \leq T(d)3^{2^d}(n+2)^{2^d}$.

LEMMA 2.12.2 *When $d \geq 4$ the number $T(d)$ of depth-d unlabeled, unoriented binary trees satisfies $T(d) \leq (56)^{2^{d-4}}$.*

Proof There is one binary tree of depth 0, a tree containing a single vertex, and one of depth 1. Let $C(d)$ be the number of unlabeled, unoriented binary trees of depth d or less, including depth 0. Thus, $C(0) = 1$, $T(1) = 1$, and $C(1) = 2$. This recurrence for $C(d)$ follows immediately for $d \geq 1$:

$$C(d) = C(d-1) + T(d) \tag{2.16}$$

We now enumerate the unoriented, unlabeled binary trees of depth $d+1$. Without loss of generality, let the left subtree of the root have depth d. There are $T(d)$ such subtrees. The right subtree can either be of depth $d-1$ or less (there are $C(d-1)$ such trees) or of depth d. In the first case there are $T(d)C(d-1)$ trees. In the second, there are $T(d)(T(d)-1)/2$ pairs of different subtrees (orientation is not counted) and $T(d)$ pairs of identical subtrees. It follows that

$$T(d+1) = T(d)C(d-1) + T(d)(T(d)-1)/2 + T(d) \tag{2.17}$$

Thus, $T(2) = 2$, $C(2) = 4$, $T(3) = 7$, $C(3) = 11$, and $T(4) = 56$. From this recurrence we conclude that $T(d+1) \geq T^2(d)/2$. We use this fact and the inequality $y \geq 1/(1-1/y)$, which holds for $y \geq 2$, to show that $(T(d+1)/T(d)) + T(d)/2 \leq T(d+1)/2$. Since $T(d) \geq 4$ for $d \geq 3$, it follows that $T(d)/2 \geq 1/(1 - 2/T(d))$. Replacing $T(d)/2$ by this lower bound in the inequality $T(d+1) \geq T^2(d)/2$, we achieve the desired result by simple algebraic manipulation. We use this fact below.

Solving the equation (2.17) for $C(d-1)$, we have

$$C(d-1) = \frac{T(d+1)}{T(d)} - \frac{(T(d)+1)}{2} \tag{2.18}$$

Substituting this expression into (2.16) yields the following recurrence:

$$\frac{T(d+2)}{T(d+1)} = \frac{T(d+1)}{T(d)} + \frac{(T(d+1)+T(d))}{2}$$

Since $(T(d+1)/T(d)) + T(d)/2 \leq T(d+1)/2$, it follows that $T(d+2)$ satisfies the inequality $T(d+2) \leq T^2(d+1)$ when $d \geq 3$ or $T(d) \leq T^2(d-1)$ when $d \geq 5$ and $d - 1 \geq 4$. Thus, $T(d) \leq T^{2^j}(d-j)$ for $d - j \geq 4$ or $T(d) \leq (56)^{2^{d-4}}$ for $d \geq 4$. ■

Combine this with the early upper bound on $N(d)$ for the number of tree circuits over Ω_0 of depth d and we have that $N(d) \leq c^{2^d}$ for $d \geq 4$, where $c = 3((56)^{1/16})(n+2)$. (Note that $3(56)^{1/16} \leq 4$.) The number of such trees of depth 0 through d is at most $N(d+1) \leq c^{2^{d+1}}$. But if $c^{2^{D_0+1}}$ is at most $2^{2^n(1-\delta)}$, then a fraction of at most $2^{-\delta 2^n}$ of the Boolean functions on n variables have depth D_0 or less. But this holds when

$$D_0 = n - 1 - \delta \log_2 e - \log_2 log_2 4(n+2) = n - \log\log n - O(1)$$

since $\ln(1-x) \leq -x$. Note that $d \geq 4$ implies that $n \geq d+1$.

THEOREM 2.12.2 *For each $0 < \delta < 1$ a fraction of at least $1 - 2^{-\delta 2^n}$ of the Boolean functions $f : \mathcal{B}^n \mapsto \mathcal{B}$ have depth complexity $D_{\Omega_0}(f)$ that satisfies the following bound when $n \geq 5$:*

$$D_{\Omega_0}(f) \geq n - \log\log n - O(1)$$

As the above two theorems demonstrate, most Boolean functions on n variables require circuits whose size and depth are approximately $2^n/n$ and n, respectively. Fortunately, most of the useful Boolean functions are far less complex than these bounds suggest. In fact, we often encounter functions whose size is polynomial in n and whose depth is logarithmic in or a small polynomial in the logarithm of the size of its input. Functions that are polynomial in the logarithm of n are called **poly-logarithmic**.

2.13 Upper Bounds on Circuit Size

In this section we demonstrate that every Boolean function on n variables can be realized with circuit size and depth that are close to the lower bounds derived in the preceding section. We begin by stating the obvious upper bounds on size and depth and then proceed to obtain stronger (that is, smaller) upper bounds on size through the use of refined arguments.

As shown in Section 2.2.2, every Boolean function $f : \mathcal{B}^n \mapsto \mathcal{B}$ can be realized as the OR of its minterms. As shown in Section 2.5.4, the minterms on n variables are produced by the decoder function $f^{(n)}_{\mathrm{decode}} : \mathcal{B}^n \mapsto \mathcal{B}^{2^n}$, which has a circuit with $2^n + (2n-2)2^{n/2}$ gates and depth $\lceil \log_2 n \rceil + 1$. Consequently, we can realize f from a circuit for $f^{(n)}_{\mathrm{decode}}$ and an OR tree on at most 2^n inputs (which has at most $2^n - 1$ two-input OR's and depth at most n). We have that every function $f : \mathcal{B}^n \mapsto \mathcal{B}$ has circuit size and depth satisfying:

$$\begin{aligned} C_\Omega(f) &\leq C_\Omega\left(f^{(n)}_{\mathrm{decode}}\right) + 2^n - 1 \leq 2^{n+1} + (2n-2)2^{n/2} - 1 \\ D_\Omega(f) &\leq D_\Omega\left(f^{(n)}_{\mathrm{decode}}\right) + n \qquad \leq n + \lceil \log_2 n + 1 \rceil + 1 \end{aligned}$$

Thus every Boolean function $f : \mathcal{B}^n \mapsto \mathcal{B}$ can be realized with an exponential number of gates and depth $n + \lceil \log_2 n \rceil + 1$. Since the depth lower bound of $n - O(\log \log n)$ applies to almost all Boolean functions on n variables (see Section 2.12), this is a very good upper bound on depth. We improve upon the circuit size bound after summarizing the depth bound.

THEOREM 2.13.1 *The depth complexity of every Boolean function $f : \mathcal{B}^n \mapsto \mathcal{B}$ satisfies the following bound:*

$$D_{\Omega_0}(f) \leq n + \lceil \log_2 n \rceil + 1$$

We now describe a procedure to construct circuits of small size for arbitrary Boolean functions on n variables. By the results of the preceding section, this size will be exponential in n. The method of approach is to view an arbitrary Boolean function $f : \mathcal{B}^n \mapsto \mathcal{B}$ on n input variables $\boldsymbol{x}$ as a function of two sets of variables, $\boldsymbol{a}$, the first k variables of $\boldsymbol{x}$, and $\boldsymbol{b}$, the remaining $n-k$ variables of $\boldsymbol{x}$. That is, $\boldsymbol{x} = \boldsymbol{ab}$ where $\boldsymbol{a} = (x_1, \dots, x_k)$ and $\boldsymbol{b} = (x_{k+1}, \dots, x_n)$.

As suggested by Fig. 2.22, we rearrange the entries in the defining table for f into a rectangular table with 2^k rows indexed by $\boldsymbol{a}$ and 2^{n-k} columns indexed by $\boldsymbol{b}$. The lower right-hand quadrant of the table contains the values of the function f. The value of f on $\boldsymbol{x}$ is the entry at the intersection of the row indexed by the value of $\boldsymbol{a}$ and the column indexed by the value of $\boldsymbol{b}$. We fix s and divide the lower right-hand quadrant of the table into $p-1$ groups of s consecutive rows and one group of $s' \leq s$ consecutive rows where $p = \lceil 2^k/s \rceil$. (Note that $(p-1)s + s' = 2^k$.) Call the ith collections of rows A_i. This table serves as the basis for the (k,s)-Lupanov representation of f, from which a smaller circuit for f can be constructed.

Let $f_i : \mathcal{B}^n \mapsto \mathcal{B}$ be f restricted to A_i; that is,

$$f_i(\boldsymbol{x}) = \begin{cases} f(\boldsymbol{x}) & \text{if } \boldsymbol{a} \in A_i \\ 0 & \text{otherwise.} \end{cases}$$

It follows that f can be expanded as the OR of the f_i:

$$f(\boldsymbol{x}) = \bigvee_{i=1}^{p} f_i(\boldsymbol{x})$$

We now expand f_i. When $\boldsymbol{b}$ is fixed, the values for $f_i(\boldsymbol{ab})$ when $\boldsymbol{a} \in A_i$ constitute an s-tuple (s'-tuple) $\boldsymbol{v}$ for $1 \leq i \leq p-1$ (for $i = p$). Let $B_{i,\boldsymbol{v}}$ be those $(n-k)$-tuples $\boldsymbol{b}$ for

		x_4	0	1	0	1	0	1	0	1	
		x_5	0	0	1	1	0	0	1	1	
		x_6	0	0	0	0	1	1	1	1	
x_1	x_2	x_3									
0	0	0	0	1	0	0	0	1	0	0	
0	0	1	0	1	1	0	0	1	1	1	A_1
0	1	0	1	0	0	1	0	0	0	1	
0	1	1	1	0	1	1	0	0	1	0	
1	0	0	0	0	0	0	1	0	0	1	A_2
1	0	1	1	1	0	1	1	0	0	0	
1	1	0	1	0	1	1	0	1	1	0	
1	1	1	0	1	0	0	0	0	1	0	A_3

Figure 2.22 The rectangular representation of the defining table of a Boolean function used in its (k, s)-Lupanov representation.

which $\boldsymbol{v}$ is the tuple of values of f_i when $\boldsymbol{a} \in A_i$. (Note that the non-empty sets $B_{i,\boldsymbol{v}}$ for different values of $\boldsymbol{v}$ are disjoint.) Let $f^{(c)}_{i,\boldsymbol{v}}(\boldsymbol{b}) : \mathcal{B}^{n-k} \mapsto \mathcal{B}$ be defined as

$$f^{(c)}_{i,\boldsymbol{v}}(\boldsymbol{b}) = \begin{cases} 1 & \text{if } \boldsymbol{b} \in B_{i,\boldsymbol{v}} \\ 0 & \text{otherwise.} \end{cases}$$

Finally, we let $f^{(r)}_{i,\boldsymbol{v}}(\boldsymbol{a}) : \mathcal{B}^k \mapsto \mathcal{B}$ be the function that has value v_j, the jth component of $\boldsymbol{v}$, when $\boldsymbol{a}$ is the jth k-tuple in A_i:

$$f^{(r)}_{i,\boldsymbol{v}}(\boldsymbol{a}) = \begin{cases} 1 & \text{if } \boldsymbol{a} \text{ is the } j\text{th element of } A_i \text{ and } v_j = 1 \\ 0 & \text{otherwise.} \end{cases}$$

It follows that $f_i(\boldsymbol{x}) = \bigvee_{\boldsymbol{v}} f^{(r)}_{i,\boldsymbol{v}}(\boldsymbol{a}) f^{(c)}_{i,\boldsymbol{v}}(\boldsymbol{b})$. Given these definitions, f can be expanded in the following **(k, s)-Lupanov representation**:

$$f(\boldsymbol{x}) = \bigvee_{i=1}^{p} \bigvee_{\boldsymbol{v}} f^{(r)}_{i,\boldsymbol{v}}(\boldsymbol{a}) \wedge f^{(c)}_{i,\boldsymbol{v}}(\boldsymbol{b}) \tag{2.19}$$

We now bound the number of logic elements needed to realize an arbitrary function $f : \mathcal{B}^n \mapsto \mathcal{B}$ in this representation.

Consider the functions $f^{(r)}_{i,\boldsymbol{v}}(\boldsymbol{a})$ for a fixed value of $\boldsymbol{v}$. We construct a decoder circuit for the minterms in $\boldsymbol{a}$ that has size at most $2^k + (k-2)2^{k/2}$. Each of the functions $f^{(r)}_{i,\boldsymbol{v}}$ can be realized as the OR of s minterms in $\boldsymbol{a}$ for $1 \leq i \leq p-1$ and s' minterms otherwise. Thus, $(p-1)(s-1) + (s'-1) \leq 2^k$ two-input OR's suffice for all values of i and a fixed value of $\boldsymbol{v}$. Hence, for each value of $\boldsymbol{v}$ the functions $f^{(r)}_{i,\boldsymbol{v}}$ can be realized by a circuit of size $O(2^k)$. Since there are at most 2^s choices for $\boldsymbol{v}$, all $f^{(r)}_{i,\boldsymbol{v}}$ can be realized by a circuit of size $O(2^{k+s})$.

Consider next the functions $f^{(c)}_{i,\boldsymbol{v}}(\boldsymbol{b})$. We construct a decoder circuit for the minterms of $\boldsymbol{b}$ that has size at most $2^{n-k} + (n-k-2)2^{(n-k)/2}$. Since for each i, $1 \leq i \leq p$, the sets

$B_{i,\boldsymbol{v}}$ for different values of $\boldsymbol{v}$ are disjoint, $f^{(c)}_{i,\boldsymbol{v}}(\boldsymbol{b})$ can be realized as the OR of at most 2^{n-k} minterms using at most 2^{n-k} two-input OR's. Thus, all $f^{(c)}_{i,\boldsymbol{v}}(\boldsymbol{b})$, $1 \leq i \leq p$, can be realized with $p2^{n-k} + 2^{n-k} + (n-k-2)2^{(n-k)/2}$ gates.

Consulting (2.19), we see that to realize f we must add one AND gate for each i and tuple $\boldsymbol{v}$. We must also add the number of two-input OR gates needed to combine these products. Since there are at most $p2^s$ products, at least $p2^s$ OR gates are needed for a total of $p2^{s+1}$ gates.

Let $C_{k,s}(f)$ be the total number of gates needed to realize f in the (k,s)-Lupanov representation. $C_{k,s}(f)$ satisfies the following inequality:

$$C_{k,s}(f) \leq O(2^{k+s}) + O(2^{(n-k)}) + p(2^{n-k} + 2^{s+1})$$

Since $p = \lceil 2^k/s \rceil$, $p \leq 2^k/s + 1$, this expands to

$$C_{k,s}(f) \leq O(2^{k+s}) + O(2^{n-k}) + \frac{2^n}{s} + \frac{2^{k+s+1}}{s}$$

Now let $k = \lceil 3 \log_2 n \rceil$ and $s = \lceil n - 5 \log_2 n \rceil$. Then, $k + s \leq n - \log_2 n^2 + 2$ and $n - k \leq n - \log_2 n^3$. As a consequence, for large n, we have

$$C_{k,s}(f) \leq O\left(\frac{2^n}{n^2}\right) + O\left(\frac{2^n}{n^3}\right) + \frac{2^n}{(n - 5\log_2 n)}$$

We summarize the result in a theorem.

THEOREM 2.13.2 *For each $\epsilon > 0$ there exists some $N_0 > 1$ such that for all $n \geq N_0$ every Boolean function $f : \mathcal{B}^n \mapsto \mathcal{B}$ has a circuit size complexity satisfying the following upper bound:*

$$C_{\Omega_0}(f) \leq \frac{2^n}{n}(1 + \epsilon)$$

Since we show in Section 2.12 that for $0 < \epsilon < 1$ almost all Boolean functions $f : \mathcal{B}^n \mapsto \mathcal{B}$ have a circuit size complexity satisfying

$$C_{\Omega_0}(f) \geq \frac{2^n}{n}(1 - \epsilon) - 2n^2$$

for $n \geq 2[(1-\epsilon)/\epsilon] \log_2[(3e)^2(1 - \epsilon/2)]$, this is a good lower bound.

Problems

MATHEMATICAL PRELIMINARIES

2.1 Show that the following identities on geometric series hold:

$$\sum_{j=0}^{s} a^j = \frac{(a^{s+1} - 1)}{(a - 1)}$$

$$\sum_{j=0}^{s} a^j j = \frac{a}{(a-1)^2}(sa^{s+1} - (s+1)a^s + 1)$$

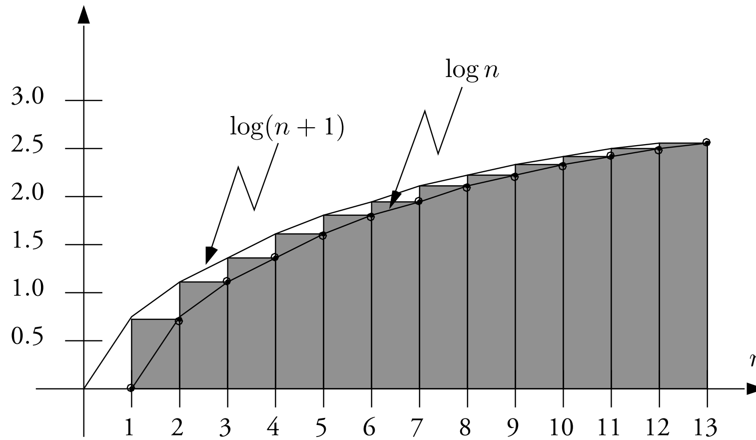


Figure 2.23 The natural logarithm of the factorial $n!$ is $\sum_{k=1}^{n} \ln k$, which is bounded below by $\int_1^n \ln x \; dx$ and above by $\int_1^n \ln(x+1) \; dx$.

2.2 Derive tight upper and lower bounds on the factorial function $n! = n(n-1)\cdots 3\,2\,1$.

Hint: Derive bounds on $\ln n!$ where $\ln$ is the natural logarithm. Use the information given in Fig. 2.23.

2.3 Let $\mathcal{T}(d)$ be a complete balanced binary tree of depth d. $\mathcal{T}(1)$, shown in Fig. 2.24(a), has a root and two leaves. $\mathcal{T}(d)$ is obtained by attaching to each of the leaves of $\mathcal{T}(1)$ copies of $\mathcal{T}(d-1)$. $\mathcal{T}(3)$ is shown in Fig. 2.24(b).

a) Show by induction that $\mathcal{T}(d)$ has 2^d leaves and $2^d - 1$ non-leaf vertices.

b) Show that any binary tree (each vertex except leaves has two descendants) with n leaves has $n-1$ non-leaf vertices and depth at least $\lceil \log_2 n \rceil$.

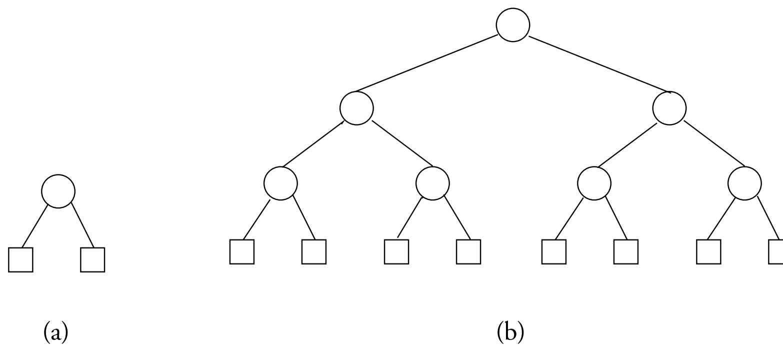


Figure 2.24 Complete balanced binary trees a) of depth one and b) depth 3.

BINARY FUNCTIONS AND LOGIC CIRCUITS

2.4 a) Write a procedure EXOR in a language of your choice that **writes** the description of the straight-line program given in equation (2.2).

b) Write a program in a language of your choice that **evaluates** an arbitrary straight-line program given in the format of equation (2.2) in which each input value is specified.

2.5 A set of Boolean functions forms a **complete basis** Ω if a logic circuit can be constructed for every Boolean function $f : \mathcal{B}^n \mapsto \mathcal{B}$ using just functions in Ω.

a) Show that the basis consisting of one function, the NAND gate, a gate on two inputs realizing the NOT of the AND of its inputs, is complete.

b) Determine whether or not the basis $\{$AND, OR$\}$ is complete.

2.6 Show that the CNF of a Boolean function f is unique and is the negation of the DNF of $\overline{f}$.

2.7 Show that the RSE of a Boolean function is unique.

2.8 Show that any SOPE (POSE) of the parity function $f_{\oplus}^{(n)}$ has exponentially many terms.
Hint: Show by contradiction that every term in a SOPE (every clause of a POSE) of $f_{\oplus}^{(n)}$ contains every variable. Then use the fact that the DNF (CNF) of $f_{\oplus}^{(n)}$ has exponentially many terms to complete the proof.

2.9 Demonstrate that the RSE of the OR of n variables, $f_{\vee}^{(n)}$, includes every product term except for the constant 1.

2.10 Consider the Boolean function $f_{\text{mod } 3}^{(n)}$ on n variables, which has value 1 when the sum of its variables is zero modulo 3 and value 0 otherwise. Show that it has exponential-size DNF, CNF, and RSE normal forms.
Hint: Use the fact that the following sum is even:

$$\sum_{0 \le j \le k} \binom{3k}{3j}$$

2.11 Show that every Boolean function $f^{(n)} : \mathcal{B}^n \mapsto \mathcal{B}$ can be expanded as follows:

$$f(x_1, x_2, \ldots, x_n) = x_1 f(1, x_2, \ldots, x_n) \vee \overline{x}_1 f(0, x_2, \ldots, x_n)$$

Apply this expansion to each variable of $f(x_1, x_2, x_3) = x_1\overline{x}_2 \vee x_2 x_3$ to obtain its DNF.

2.12 In a **dual-rail logic** circuit 0 and 1 are represented by the pairs $(0, 1)$ and $(1, 0)$, respectively. A variable x is represented by the pair $(x, \overline{x})$. A NOT in this representation (called a DRL-NOT) is a pair of twisted wires.

a) How are AND (DRL-AND) and OR (DRL-OR) realized in this representation? Use standard AND and OR gates to construct circuits for gates in the new representation. Show that every function $f : \mathcal{B}^n \mapsto \mathcal{B}^m$ can be realized by a **dual-rail logic** circuit in which the standard NOT gates are used only on input variables (to obtain the pair $(x, \overline{x})$).

b) Show that the size and depth of a dual-rail logic circuit for a function $f : \mathcal{B}^n \mapsto \mathcal{B}$ are at most twice the circuit size (plus the NOTs for the inputs) and at most one more than the circuit depth of f over the basis {AND, OR, NOT}, respectively.

2.13 A function $f : \mathcal{B}^n \mapsto \mathcal{B}$ is **monotone** if for all $1 \leq j \leq n$, $f(x_1, \dots, x_{j-1}, 0, x_{j+1}, \dots, x_n) \leq f(x_1, \dots, x_{j-1}, 1, x_{j+1}, \dots, x_n)$ for all values of the remaining variables; that is, increasing any variable from 0 to 1 does not cause the function to decrease its value from 1 to 0.

a) Show that every circuit over the basis $\Omega_{\text{mon}} = \{\text{AND}, \text{OR}\}$ computes monotone functions at every gate.

b) Show that every monotone function $f^{(n)} : \mathcal{B}^n \mapsto \mathcal{B}$ can be expanded as follows:

$$f(x_1, x_2, \dots, x_n) = x_1 f(1, x_2, \dots, x_n) \vee f(0, x_2, \dots, x_n)$$

Show that this implies that every monotone function can be realized by a logic circuit over the **monotone basis** $\Omega_{\text{mon}} = \{\text{AND}, \text{OR}\}$.

SPECIALIZED FUNCTIONS

2.14 Complete the proof of Lemma 2.5.3 by solving the recurrences stated in Equation (2.4).

2.15 Design a multiplexer circuit of circuit size 2^{n+1} plus lower-order terms when n is even.

Hint: Construct a smaller circuit by applying the decomposition given in Section 2.5.4 of the minterms of n variables into minterms on the two halves of the n variables.

2.16 Complete the proof of Lemma 2.11.1 by establishing the correctness of the inductive hypothesis stated in its proof.

2.17 The **binary sorting** function is defined in Section 2.11. Show that it can be realized with a circuit whose size is $O(n)$ and depth is $O(\log n)$.

Hint: Consider using a circuit for $f^{(m)}_{\text{count}}$, a decoder circuit and other circuitry. Is there a role for a prefix computation in this problem?

LOGICAL FUNCTIONS

2.18 Let $f^{(n)}_{\text{member}} : \mathcal{B}^{(n+1)b} \mapsto \mathcal{B}$ be defined below.

$$f^{(n)}_{\text{member}}(\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_n, \boldsymbol{y}) = \begin{cases} 1 & \boldsymbol{x}_i = \boldsymbol{y} \quad \text{for some } 1 \leq i \leq n \\ 0 & \text{otherwise} \end{cases}$$

where $\boldsymbol{x}_i, \boldsymbol{y} \in \mathcal{B}^b$ and $\boldsymbol{x}_i = \boldsymbol{y}$ if and only if they agree in each position.

Obtain good upper bounds to $C_\Omega\left(f^{(n)}_{\text{member}}\right)$ and $D_\Omega\left(f^{(n)}_{\text{member}}\right)$ by constructing a circuit over the basis $\Omega = \{\wedge, \vee, \neg, \oplus\}$.

2.19 Design a circuit to compare two n-bit binary numbers and return the value 1 if the first is larger than or equal to the second and 0 otherwise.

Hint: Compare each pair of digits of the same significance and generate three outcomes, **yes**, **maybe**, and **no**, corresponding to whether the first digit is greater than, equal to or less than the second. How can you combine the outputs of such a comparison circuit to design a circuit for the problem? Does a prefix computation appear in your circuit?

PARALLEL PREFIX

2.20 a) Let $\odot_{\text{copy}} : S^2 \mapsto S$ be the operation

$$a \odot_{\text{copy}} b = a$$

Show that $(S, \odot_{\text{copy}})$ is a semigroup for S an arbitrary non-empty set.

b) Let $\cdot$ denote string concatenation over the set $\{0, 1\}^*$ of binary strings. Show that it is associative.

2.21 The segmented prefix computation with the associative operation $\odot$ on a "value" n-vector $\boldsymbol{x}$ over a set S, given a "flag vector" $\boldsymbol{\phi}$ over $\mathcal{B}$, is defined as follows: the value of the ith entry y_i of the "result vector" $\boldsymbol{y}$ is x_i if its flag is 1 and otherwise is the associative combination with $\odot$ of x_i and the entries to its left up to and including the first occurrence of a 1 in the flag array. The leftmost bit in every flag vector is 1. An example of a segmented prefix computation is given in Section 2.6.

Assuming that $(S, \odot)$ is a semigroup, a segmented prefix computation over the set $S \times \mathcal{B}$ of pairs is a special case of general prefix computation. Consider the operator $\otimes$ on pairs (x_i, ϕ_i) of values and flags defined below:

$$((x_1, \phi_1) \otimes (x_2, \phi_2)) = \begin{cases} (x_2, 1) & \phi_2 = 1 \\ (x_1 \odot x_2, \phi_1) & \phi_2 = 0 \end{cases}$$

Show that $((S, \mathcal{B}), \otimes)$ is a semigroup by proving that $(S, \mathcal{B})$ is closed under the operator $\otimes$ and that the operator $\otimes$ is associative.

2.22 Construct a logic circuit of size $O(n \log n)$ and depth $O(\log^2 n)$ that, given a binary n-tuple $\boldsymbol{x}$, computes the n-tuple $\boldsymbol{y}$ containing the running sum of the number of 1's in $\boldsymbol{x}$.

2.23 Given $2n$ Boolean variables organized as pairs $0a$ or $1a$, design a circuit that moves pairs of the form $1a$ to the left and the others to the right without changing their relative order. Show that the circuit has size $O(n \log^2 n)$.

2.24 Linear recurrences play an important role in many problems including the solution of a tridiagonal linear system of equations. They are defined over "near-rings," which are slightly weaker than rings in not requiring inverses under the addition operation. (Rings are defined in Section 6.2.1.)

A **near-ring** $(\mathcal{R}, \cdot, +)$ is a set $\mathcal{R}$ together with an associative multiplication operator $\cdot$ and an associative and commutative addition operator $+$. (If $+$ is commutative, then for all $a, b \in \mathcal{R}$, $a + b = b + a$.) In addition, $\cdot$ distributes over $+$; that is, for all $a, b, c \in \mathcal{R}$, $a \cdot (b + c) = a \cdot b + a \cdot c$.

A **first-order linear recurrence of length** n is an n-tuple $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ of variables over a near-ring $(\mathcal{R}, \cdot, +)$ that satisfies $x_1 = b_1$ and the following set of identities for $2 \le j \le n$ defined in terms of elements $\{a_j, b_j \in \mathcal{R} \mid\ 2 \le j \le n\}$:

$$x_j = a_j \cdot x_{j-1} + b_j$$

Use the ideas of Section 2.7 on carry-lookahead addition to show that x_j can be written

$$x_j = c_j \cdot x_1 + d_j$$

where the pairs (c_j, d_j) are the result of a prefix computation.

ARITHMETIC OPERATIONS

2.25 Design a circuit that finds the most significant non-zero position in an n-bit binary number and logically shifts the binary number left so that the non-zero bit is in the most significant position. The circuit should produce not only the shifted binary number but also a binary representation of the amount of the shift.

2.26 Consider the function $\pi[j,k] = \pi[j,k-1] \diamond \pi[k,k]$ for $1 \leq j < k \leq n-1$, where $\diamond$ is defined in Section 2.7.1. Show by induction that the first component of $\pi[j,k]$ is 1 if and only if a carry propagates through the full adder stages numbered $j, j+1, \dots, k$ and its second component is 1 if and only if a carry is generated at one of these stages, propagates through subsequent stages, and appears as a carry out of the kth stage.

2.27 Give a construction of a circuit for subtracting one n-bit positive binary integer from another using the two's-complement operation. Show that the circuit has size $O(n)$ and depth $O(\log n)$.

2.28 Complete the proof of Theorem 2.9.3 outlined in the text. In particular, solve the recurrence given in equation (2.10).

2.29 Show that the depth bound stated in Theorem 2.9.3 can be improved from $O(\log^2 n)$ to $O(\log n)$ without affecting the size bound by using carry-save addition to form the six additions (or subtractions) that are involved at each stage.

Hint: Observe that each multiplication of $(n/2)$-bit numbers at the top level is expanded at the next level as sums of the product of $(n/4)$-bit numbers and that this type of replacement continues until the product is formed of 1-bit numbers. Observe also that $2n$-bit carry-save adders can be used at the top level but that the smaller carry-save adders can be used at successively lower levels.

2.30 Residue arithmetic can be used to add and subtract integers. Given positive relatively prime integers $p_1, p_2, \dots, p_k$ (no common factors), an integer n in the set $\{0, 1, 2, \dots, N-1\}$, $N = p_1 p_2 \cdots p_k$, can be represented by the k -tuple $\boldsymbol{n} = (n_1, n_2, \dots, n_k)$, where $n_j = n \bmod p_j$. Let n and m be in this set.

a) Show that if $n \neq m$, $\boldsymbol{n} \neq \boldsymbol{m}$.

b) Form $\boldsymbol{n} + \boldsymbol{m}$ by adding corresponding jth components modulo p_j. Show that $\boldsymbol{n} + \boldsymbol{m}$ uniquely represents $(n+m) \bmod N$.

c) Form $n \times m$ by multiplying corresponding jth components of $\boldsymbol{n}$ and $\boldsymbol{m}$ modulo p_j. Show that $n \times m$ is the unique representation for $(nm) \bmod N$.

2.31 Use the circuit designed in Problem 2.19 to build a circuit that adds two n-bit binary numbers modulo an arbitrary third n-bit binary number. You may use known circuits.

2.32 In **prime factorization** an integer n is represented as the product of primes. Let $p(N)$ be the largest prime less than N. Then, $n \in \{2, \dots, N-1\}$ is represented by the exponents $(e_2, e_3, \dots, e_{p(N)})$, where $n = 2^{e_2} 3^{e_3} \dots p(N)^{e_{p(N)}}$. The representation for the product of two integers in this system is the sum of the exponents of their respective prime factors. Show that this leads to a multiplication circuit whose depth is proportional to $\log \log \log N$. Determine the size of the circuit using the fact that there are $O(N/\log N)$ primes in the set $\{2, \dots, N-1\}$.

2.33 Construct a circuit for the division of two n-bit binary numbers from circuits for the reciprocal function $f_{\mathrm{recip}}^{(n)}$ and the integer multiplication function $f_{\mathrm{mult}}^{(n)}$. Determine the size and depth of this circuit and the accuracy of the result.

2.34 Let $f : \mathcal{B}^n \mapsto \mathcal{B}^{kn}$ be an integer power of $\boldsymbol{x}$; that is, $f(\boldsymbol{x}) = \boldsymbol{x}^k$ for some integer k. Show that such functions contain the shifting function $f_{\mathrm{shift}}^{(m)}$ as a subfunction for some integer m. Determine m dependent on n and k.

2.35 Let $f : \mathcal{B}^n \mapsto \mathcal{B}^n$ be a fractional power of $\boldsymbol{x}$ of the form $f(\boldsymbol{x}) = \lceil \boldsymbol{x}^{q/2^k} \rceil$, $0 < q < 2^k < \log_2 n$. Show that this function contains the shifting function $f_{\mathrm{shift}}^{(m)}$ as a subfunction. Find the largest value of m for which this holds.

Chapter Notes

Logic circuits have a long history. Early in the nineteenth century Babbage designed mechanical computers capable of logic operations. In the twentieth century logic circuits, called switching circuits, were constructed of electromechanical relays. The earliest formal analysis of logic circuits is attributed to Claude Shannon [305]; he applied Boolean algebra to the analysis of logic circuits, the topic of Section 2.2. Reduction between problems, a technique central to computer science, is encountered whenever one uses an existing program to solve a new problem by pre-processing inputs and post-processing outputs. Reductions also provide a way to identify problems with similar complexity, an idea given great importance by the work of Cook [74], Karp [158], and Levin [198] on **NP**-completeness. (See also [334].) This topic is explored in depth in Chapter 8.

The upper bound on the size of ripple adder described in Section 2.7 cannot be improved, as shown by Red'kin [275] using the gate elimination method of Section 9.3.2. Prefix computations, the subject of Section 2.6, were first used by Ofman [233]. He constructed the adder based on carry-lookahead addition described in Section 2.7. Krapchenko [172] and Brent [57] developed adders with linear size whose depth is $\lceil \log n \rceil + O(\sqrt{\lceil \log n \rceil})$, asymptotically almost as good at the best possible depth bound of $\lceil \log n \rceil$.

Ofman used carry-save addition for fast integer multiplication [233]. Wallace independently discovered carry-save addition and logarithmic depth circuits for addition and multiplication [355]. The divide-and-conquer integer multiplication algorithm of Section 2.9.2 is due to Karatsuba [154]. As mentioned at the end of Section 2.9, Schönhage and Strassen [302] have designed binary integer multipliers of depth $O(\log n)$ whose size is $O(n \log n \log \log n)$.

Sir Isaac Newton around 1665 invented the iterative method bearing his name used in Section 2.10 for binary integer division. Our treatment of this idea follows that given by Tate [324]. Reif and Tate [277] have shown that binary integer division can be done with circuit size $O(n \log n \log \log n)$ and depth $O(\log n \log \log n)$ using circuits whose description is log-space uniform. Beame, Cook, and Hoover [33] have given an $O(\log n)$-depth circuit for the reciprocal function, the best possible depth bound up to a constant multiple, but one whose size is polynomial in n and whose description is not uniform; it requires knowledge of about $n^2 / \log n$ primes.

The key result in Section 2.11 on symmetric functions is due to Muller and Preparata [225]. As indicated, it is the basis for showing that every one-output symmetric function can be realized by a circuit whose size and depth are linear and logarithmic, respectively.

Shannon [306] developed lower bounds for two-terminal switching circuits of the type given in Section 2.12 on circuit size. Muller [223] extended the techniques of Shannon to derive the lower bounds on circuit size given in Theorem 2.12.1. Shannon and Riordan [280] developed a lower bound of $\Omega(2^n/\log n)$ on the size of Boolean formulas, circuits in which the fan-out of each gate is 1. As seen in Chapter 9, such bounds readily translate into lower bounds on depth of the form given Theorem 2.12.2. Gaskov, using the Lupanov representation, has derived a comparable upper bound [109].

The upper bound on circuit size given in Section 2.13 is due to Lupanov [207]. Shannon and Riordan [280] show that a lower bound of $\Omega(2^n/\log n)$ must apply to the formula size (see Definition 9.1.1) of most Boolean functions on n variables. Given the relationship of Theorem 9.2.2 between formula size and depth, a depth lower bound of $n - \log\log n - O(1)$ follows.

Early work on circuits and circuit complexity is surveyed by Paterson [236] and covered in depth by Savage [286]. More recent coverage of this subject is contained in the survey article by Bopanna and Sipser [50] and books by Wegener [359] and Dunne [91].