CHAPTER 13

# Using Logic to Design Computer Components

In this chapter we shall see that the propositional logic studied in the previous chapter can be used to design *digital* electronic circuits. Such circuits, found in every computer, use two voltage levels ("high" and "low") to represent the binary values 1 and 0. In addition to gaining some appreciation for the design process, we shall see that algorithm-design techniques, such as "divide-and-conquer," can also be applied to hardware. In fact, it is important to realize that the process of designing a digital circuit to perform a given logical function is quite similar in spirit to the process of designing a computer program to perform a given task. The data models differ significantly, and frequently circuits are designed to do many things *in parallel* (at the same time) while common programming languages are designed to execute their steps *sequentially* (one at a time). However, general programming techniques like modularizing a design are as applicable to circuits as they are to programs.

**Parallel and sequential operation**

## 13.1 What This Chapter is About

This chapter covers the following concepts from digital circuit design:

- The notion of a gate, an electronic circuit that performs a logical operation (Section 13.2).
- How gates are organized into circuits (Section 13.3).
- Certain kinds of circuits, called combinational, that are an electronic equivalent of logical expressions (Section 13.4).
- Physical constraints under which circuits are designed, and what properties circuits must have to produce their answers quickly (Section 13.5).

- Two interesting examples of circuits: adders and multiplexers. Sections 13.6 and 13.7 show how a fast circuit can be designed for each problem using a divide-and-conquer technique.
- The memory element as an example of a circuit that remembers its input. In contrast, a combinational circuit cannot remember inputs received in the past (Section 13.8).

## 13.2 Gates

A *gate* is an electronic device with one or more inputs, each of which can assume either the value 0 or the value 1. As mentioned earlier, the logical values 0 and 1 are generally represented electronically by two different voltage levels, but the physical method of representation need not concern us. A gate usually has one output, which is a function of its inputs, and which is also either 0 or 1.

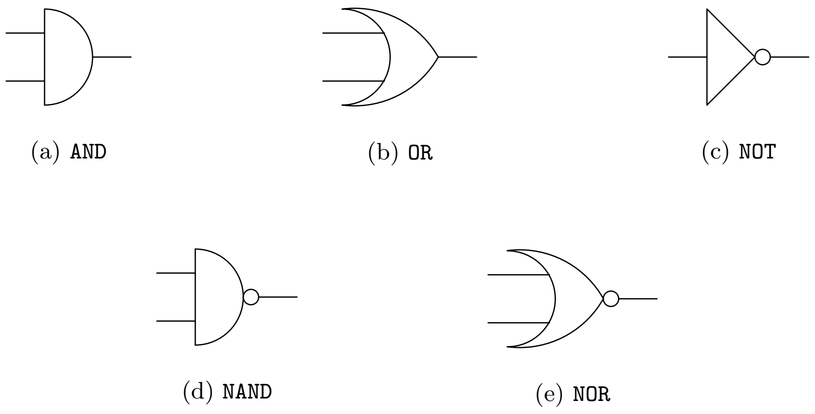


**Fig. 13.1.** Symbols for gates.

Each gate computes some particular Boolean function. Most electronic "technologies" (ways of manufacturing electronic circuits) favor the construction of gates for certain Boolean functions and not others. In particular, AND- and OR-gates are usually easy to build, as are NOT-gates, which are called *inverters.* AND- and OR-gates can have any number of inputs, although, as we discuss in Section 13.5, there is usually a practical limitation on how many inputs a gate can have. The output of an AND-gate is 1 if all its inputs are 1, and its output is 0 if any one or more of its inputs are 0. Likewise, the output of an OR-gate is 1 if one or more of its inputs are 1, and the output is 0 if all inputs are 0. The inverter (NOT-gate) has one input; its output is 1 if its input is 0 and 0 if its input is 1.

**Inverter**

We also find it easy to implement NAND- and NOR-gates in most technologies. The NAND-gate produces the output 1 unless all its inputs are 1, in which case it produces the output 0. The NOR-gate produces the output 1 when all inputs are 0 and produces 0 otherwise. An example of a logical function that is harder to implement electronically is equivalence, which takes two inputs $x$ and $y$ and produces a 1 output if $x$ and $y$ are both 1 or both 0, and a 0 output when exactly one of $x$ and $y$ is 1. However, we can build equivalence circuits out of `AND`-, `OR`-, and `NOT`-gates by implementing a circuit that realizes the logical function $xy + \bar{x}\bar{y}$.

The symbols for the gates we have mentioned are shown in Fig. 13.1. In each case except for the inverter (`NOT`-gate), we have shown the gate with two inputs. However, we could easily show more than two inputs, by adding additional lines. A one-input `AND`- or `OR`-gate is possible, but doesn't really do anything; it just passes its input to the output. A one-input `NAND`- or `NOR`-gate is really an inverter.

## ✣ 13.3 Circuits

**Circuit inputs and outputs**

Gates are combined into circuits by connecting the outputs of some gates to the inputs of others. The circuit as a whole has one or more inputs, each of which can be inputs to various gates within the circuit. The outputs of one or more gates are designated circuit outputs. If there is more than one output, then an order for the output gates must be specified as well.

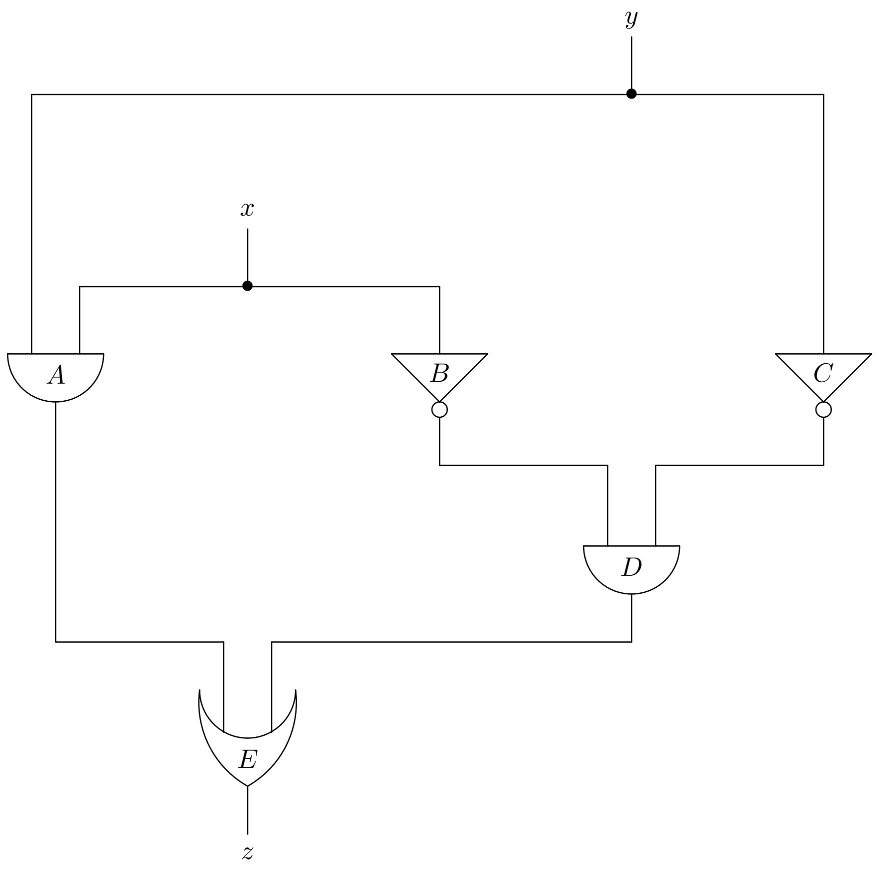


**Fig. 13.2.** Equivalence circuit: $z$ is the expression $x \equiv y$.

✦ **Example 13.1.** Figure 13.2 shows a circuit that produces as output $z$, the equivalence function of inputs $x$ and $y$. Conventionally, we show inputs at the top. Both inputs $x$ and $y$ are fed to gate $A$, which is an `AND`-gate, and which therefore produces a 1 output when (and only when) $x = y = 1$. Also, $x$ and $y$ are inverted by `NOT`-gates $B$ and $C$ respectively, and the outputs of these inverters are fed to `AND`-gate $D$. Thus, the output of gate $D$ is 1 if and only if both $x$ and $y$ are 0. Since the outputs of gates $A$ and $D$ are fed to `OR`-gate $E$, we see that the output of that gate is 1 if and only if either $x = y = 1$ or $x = y = 0$. The table in Fig. 13.3 gives a logical expression for the output of each gate.

Thus, the output $z$ of the circuit, which is the output of gate $E$, is 1 if and only if the logical expression $xy + \bar{x}\bar{y}$ has value 1. Since this expression is equivalent to the expression $x \equiv y$, we see that the circuit output is the equivalence function of its two inputs. ✦

| Gate | Output of Gate |
|---|---|
| $A$ | $xy$ |
| $B$ | $\bar{x}$ |
| $C$ | $\bar{y}$ |
| $D$ | $\bar{x}\bar{y}$ |
| $E$ | $xy + \bar{x}\bar{y}$ |

**Fig. 13.3.** Outputs of gates in Fig. 13.2.

## Combinational and Sequential Circuits

There is a close relationship between the logical expressions we can write using a collection of logical operators, such as `AND`, `OR`, and `NOT`, on one hand, and the circuits built from gates that perform the same set of operators, on the other hand. Before proceeding, we must focus our attention on an important class of circuits called *combinational circuits*. These circuits are acyclic, in the sense that the output of a gate cannot reach its input, even through a series of intermediate gates.

We can use our knowledge of graphs to define precisely what we mean by a combinational circuit. First, draw a directed graph whose nodes correspond to the gates of the circuit. Add an arc $u \to v$ if the output of gate $u$ is connected directly to any input of gate $v$. If the circuit's graph has no cycles, then the circuit is *combinational*; otherwise, it is *sequential*.

✦ **Example 13.2.** In Fig. 13.4 we see the directed graph that comes from the circuit of Fig. 13.2. For example, there is an arc $A \to E$ because the output of gate $A$ is connected to an input of gate $E$. The graph of Fig. 13.4 clearly has no cycles; in fact, it is a tree with root $E$, drawn upside-down. Thus, we conclude that the circuit of Fig. 13.2 is combinational.

On the other hand, consider the circuit of Fig. 13.5(a). There, the output of gate $A$ is an input to gate $B$, and the output of $B$ is an input to $A$. The graph for this circuit is shown in Fig. 13.5(b). It clearly has a cycle, so that the circuit is sequential.

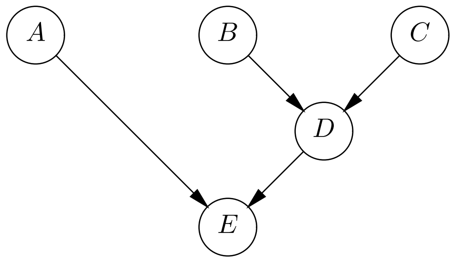
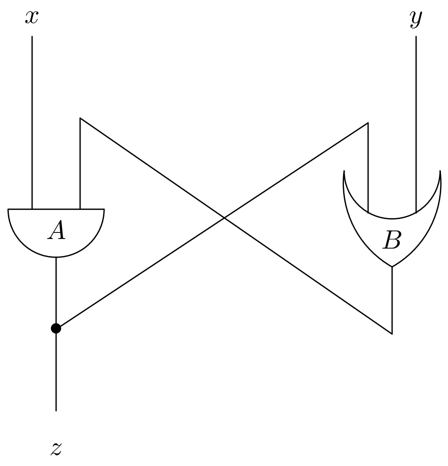
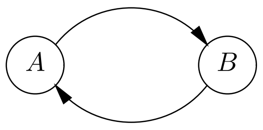


**Fig. 13.4.** Directed graph constructed from the circuit of Fig. 13.2.



(a) The circuit.



(b) Its graph.

**Fig. 13.5.** Sequential circuit and its graph.

Suppose inputs $x$ and $y$ to this circuit are both 1. Then the output of $B$ is surely 1, and therefore, both inputs to the `AND`-gate $A$ are 1. Thus, this gate will produce output 1. Now we can let input $y$ become 0, and the output of `OR`-gate $B$ will remain 1, because its other input (the input from the output of $A$) is 1. Thus, both inputs to $A$ remain 1, and its output is 1 as well.

However, suppose $x$ becomes 0, whether or not $y$ is 0. Then the output of gate $A$, and therefore the circuit output $z$, must be 0. We can describe the circuit output $z$ as 1 if, at some time in the past, both $x$ and $y$ were 1 and since then $x$ (but not necessarily $y$) has remained 1. Figure 13.6 shows the output as a function of time for various input value combinations; the low level represents 0 and the elevated

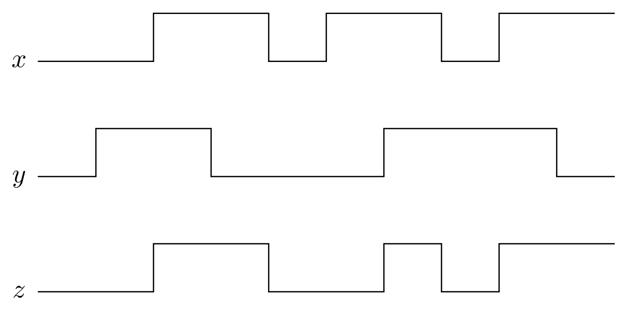


**Fig. 13.6.** Output as a function of time, for the circuit of Fig. 13.5(a).

## Sequential Circuits and Automata

There is a close relationship between the deterministic finite automata that we discussed in Chapter 10 and sequential circuits. While the subject is beyond the scope of this book, given any deterministic automaton, we can design a sequential circuit whose output is 1 exactly when the sequence of inputs of the automaton is accepted. To be more precise, the inputs of the automaton, which may be from any set of characters, must be encoded by the appropriate number of logical inputs (which each take the value 0 or 1); $k$ logical inputs to the circuit can code up to $2^k$ characters.

level represents 1. ✦

We shall discuss sequential circuits briefly at the end of this chapter. As we just saw in Example 13.2, sequential circuits have the ability to remember important things about the sequence of inputs seen so far, and thus they are needed for key components of computers, such as main memory and registers. Combinational circuits, on the other hand, can compute the values of logical functions, but they must work from a single setting for their inputs, and cannot remember what the inputs were set to previously. Nevertheless, combinational circuits are also vital components of computers. They are needed to add numbers, decode instructions into the electronic signals that cause the computer to perform those instructions, and many other tasks. In the following sections, we shall devote most of our attention to the design of combinational circuits.

## EXERCISES

**13.3.1**: Design circuits that produce the following outputs. You may use any of the gates shown in Fig. 13.1.

**Parity function**

a) The *parity*, or sum-mod-2, function of inputs $x$ and $y$ that is 1 if and only if exactly one of $x$ and $y$ is 1.

**Majority function**

b) The *majority* function of inputs $w$, $x$, $y$, and $z$ that is 1 if and only if three or more of the inputs are 1.

c) The function of inputs $w$, $x$, $y$, and $z$ that is 1 unless all or none of the inputs are 1.

d) The exclusive-or function $\oplus$ discussed in Exercise 12.4.7.

**13.3.2***: Suppose the circuit of Fig. 13.5(a) is modified so that both gates $A$ and $B$ are AND-gates, and both inputs $x$ and $y$ are initially 1. As the inputs change, under what circumstances will the output be 1?

**13.3.3***: Repeat Exercise 13.3.2 if both gates are OR-gates.

## 13.4 Logical Expressions and Circuits

It is relatively simple to build a circuit whose output, as a function of its inputs, is the same as that of a given logical expression. Conversely, given a combinational circuit, we can find a logical expression for each circuit output, as a function of its inputs. The same is not true of a sequential circuit, as we saw in Example 13.2.

### From Expressions to Circuits

Given a logical expression with some set of logical operators, we can construct from it a combinational circuit that uses gates with the same set of operators and realizes the same Boolean function. The circuit we construct will always have the form of a tree. We construct the circuit by a structural induction on the expression tree for the expression.

**BASIS.** If the expression tree is a single node, the expression can only be an input, say $x$. The "circuit" for this expression will be the circuit input $x$ itself.

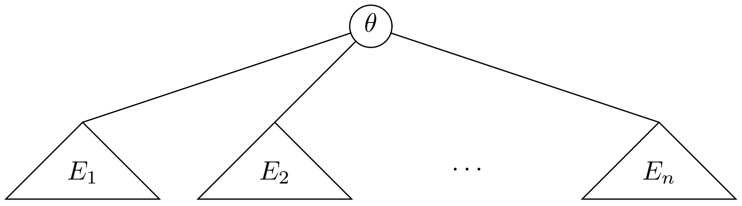


**Fig. 13.7.** Expression tree for expression $\theta(E_1, E_2, \ldots, E_n)$.

**INDUCTION.** For the induction, suppose that the expression tree in question is similar to Fig. 13.7. There is some logical operator, which we call $\theta$, at the root; $\theta$ might be AND or OR, for example. The root has $n$ subtrees for some $n$, and the operator $\theta$ is applied to the results of these subtrees to produce a result for the whole tree.

Since we are performing a structural induction, we may assume that the inductive hypothesis applies to the subexpressions. Thus, there is a circuit $C_1$ for expression $E_1$, circuit $C_2$ for $E_2$, and so on.

To build the circuit for $E$, we take a gate for the operator $\theta$ and give it $n$ inputs, one from each of the outputs of the circuits $C_1, C_2, \ldots, C_n$, in that order.

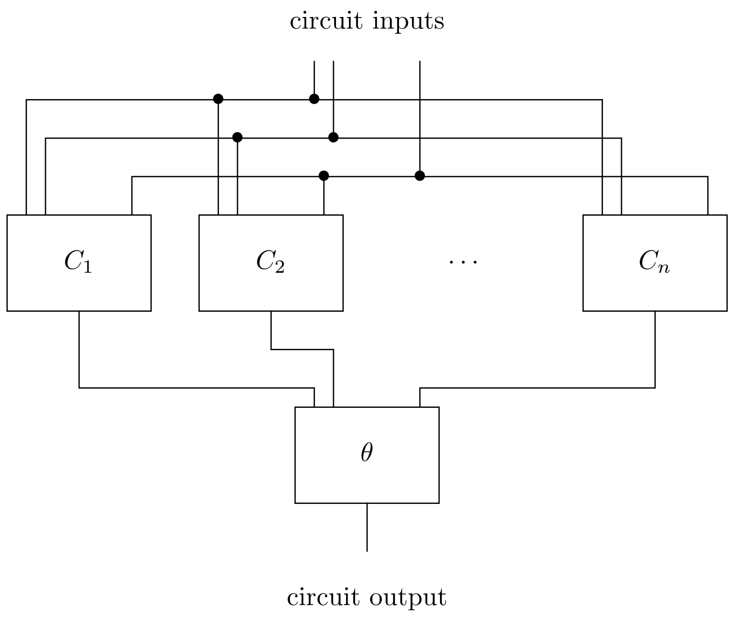


**Fig. 13.8.** The circuit for $\theta(E_1, \ldots, E_n)$ where $C_i$ is the circuit for $E_i$.

The output of the circuit for $E$ is taken from the $\theta$-gate just introduced. The construction is suggested in Fig. 13.8.

The circuit we have constructed computes the expression in the obvious way. However, there may be circuits producing the same output function with fewer gates or fewer levels. For example, if the given expression is $(x + y)z + (x + y)\bar{w}$, then the circuit we construct will have two occurrences of the subcircuit that realizes the common expression $x + y$. We can redesign the circuit to use just one occurrence of this subcircuit, and feed its output everywhere the common subexpression is used.

There are other more radical transformations that we can make to improve the design of circuits. Circuit design, like the design of efficient algorithms, is an art, and we shall see a few of the important techniques of this art later in this chapter.

## From Circuits to Logical Expressions

Now let us consider the inverse problem, constructing a logical expression for an output of a combinational circuit. Since we know that the graph of the circuit is acyclic, we can pick a topological order of its nodes (i.e., of its gates), with the property that if the output of the $i$th gate in the order feeds an input of the $j$th gate in the order, then $i$ must be less than $j$.

✦ **Example 13.3.** One possible topological order of the gates in the circuit of Fig. 13.2 is $ABCDE$, and another is $BCDAE$. However, $ABDCE$ is not a topological order, since gate $C$ feeds gate $D$, but $D$ appears before $C$ in this sequence. ✦

To build the expression from the circuit, we use an inductive construction. We shall show by induction on $i$ the statement

**STATEMENT** $S(i)$: For the first $i$ gates in the topological order, there are logical expressions for the output of these gates.

**BASIS.** The basis will be $i = 0$. Since there are zero gates to consider, there is nothing to prove, so the basis part is done.

**INDUCTION.** For the induction, look at the $i$th gate in the topological order. Suppose gate $i$'s inputs are $I_1, I_2, \ldots, I_k$. If $I_j$ is a circuit input, say $x$, then let the expression $E_j$ for input $I_j$ be $x$. If input $I_j$ is the output of some other gate, that gate must precede the $i$th gate in the topological order, which means that we have already constructed some expression $E_j$ for the output of that gate. Let the operator associated with gate $i$ be $\theta$. Then an expression for gate $i$ is $\theta(E_1, E_2, \ldots, E_k)$. In the common case that $\theta$ is a binary operator for which infix notation is conventionally used, the expression for gate $i$ can be written $(E_1)\theta(E_2)$. The parentheses are placed there for safety, although depending on the precedence of operators, they may or may not be necessary.

✦ **Example 13.4.** Let us determine the output expression for the circuit in Fig. 13.2, using the topological order $ABCDE$ for the gates. First, we look at `AND`-gate $A$. Its two inputs are from the circuit inputs $x$ and $y$, so that the expression for the output of $A$ is $xy$.

Gate $B$ is an inverter with input $x$, so that its output is $\bar{x}$. Similarly, gate $C$ has output expression $\bar{y}$. Now we can work on gate $D$, which is an `AND`-gate with inputs taken from the outputs of $B$ and $C$. Thus, the expression for the output of $D$ is $\bar{x}\bar{y}$. Finally, gate $E$ is an `OR`-gate, whose inputs are the outputs of $A$ and $D$. We thus connect the output expressions for these gates by the `OR` operator, to get the expression $xy + \bar{x}\bar{y}$ as the output expression for gate $E$. Since $E$ is the only output gate of the circuit, that expression is also the circuit output. Recall that the circuit of Fig. 13.2 was designed to realize the Boolean function $x \equiv y$. It is easy to verify that the expression we derived for gate $E$ is equivalent to $x \equiv y$. ✦

✦ **Example 13.5.** In the previous examples, we have had only one circuit output, and the circuit itself has been a tree. Neither of these conditions holds generally. We shall now take up an important example of the design of a circuit with multiple outputs, and where some gates have their output used as input to several gates.

**One-bit adder**

Recall from Chapter 1 that we discussed the use of a *one-bit adder* in building a circuit to add binary numbers. A one-bit adder circuit has two inputs $x$ and $y$ that represent the bits in some particular position of the two numbers being added. It has a third input, $c$, that represents the carry-in to this position from the position to the right (next lower-order position). The one-bit adder produces as output the following two bits:

1. The *sum bit* $z$, which is 1 if an odd number of $x$, $y$, and $c$ are 1, and
2. The *carry-out* bit $d$, which is 1 if two or more of $x$, $y$, and $c$ are 1.

## Circuit Diagram Convention

When circuits are complicated, as is the circuit in Fig. 13.10, there is a useful convention that helps simplify the drawing. Often, we need to have "wires" (the lines between an output and the input(s) to which it is connected) cross, without implying that they are part of the same wire. Thus, the standard convention for circuits says that wires are not connected unless, at the point of intersection, we place a dot. For example, the vertical line from the circuit input $y$ is not connected to the horizontal lines labeled $x$ or $\bar{x}$, even though it crosses those lines. It is connected to the horizontal line labeled $y$, because there is a dot at the point of intersection.

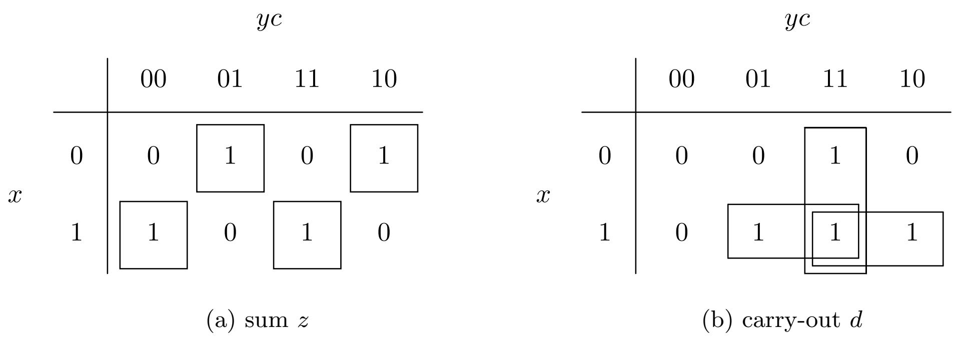


**Fig. 13.9.** Karnaugh maps for the sum and carry-out functions.

In Fig. 13.9 we see Karnaugh maps for $z$ and $d$, the sum and carry-out functions of the one-bit adder. Of the eight possible minterms, seven appear in the functions for $z$ or $d$, and only one, $xyc$, appears in both.

A systematically designed circuit for the one-bit adder is shown in Fig. 13.10. We begin by taking the circuit inputs and inverting them, using the three inverters at the top. Then we create AND-gates for each of the minterms that we need in one or more outputs. These gates are numbered 1 through 7, and each integer tells us which of its inputs are "true" circuit inputs, $x$, $y$, or $c$, and which are "complemented" inputs, $\bar{x}$, $\bar{y}$, or $\bar{c}$. That is, write the integer as a 3-bit binary number, and regard the bits as representing $x$, $y$, and $c$, in that order. For example, gate 4, or $(100)_2$, has input $x$ true and inputs $y$ and $c$ complemented; that is, it produces the output expression $x\bar{y}\bar{c}$. Notice that there is no gate 0 here, because the minterm $\bar{x}\bar{y}\bar{c}$ is not needed for either output.

Finally, the circuit outputs, $z$ and $d$, are assembled with OR-gates at the bottom. The OR-gate for $z$ has inputs from the output of each AND-gate whose minterm makes $z$ true, and the inputs to the OR-gate for $d$ are selected similarly.

Let us compute the output expressions for the circuit of Fig. 13.10. The topological order we shall use is the inverters first, then the AND-gates $1, 2, \ldots, 7$, and finally the OR-gates for $z$ and $d$. First, the three inverters obviously have output expressions $\bar{x}$, $\bar{y}$, and $\bar{c}$. Then we already mentioned how the inputs to the AND-gates were selected and how the expression for the output of each is associated with the

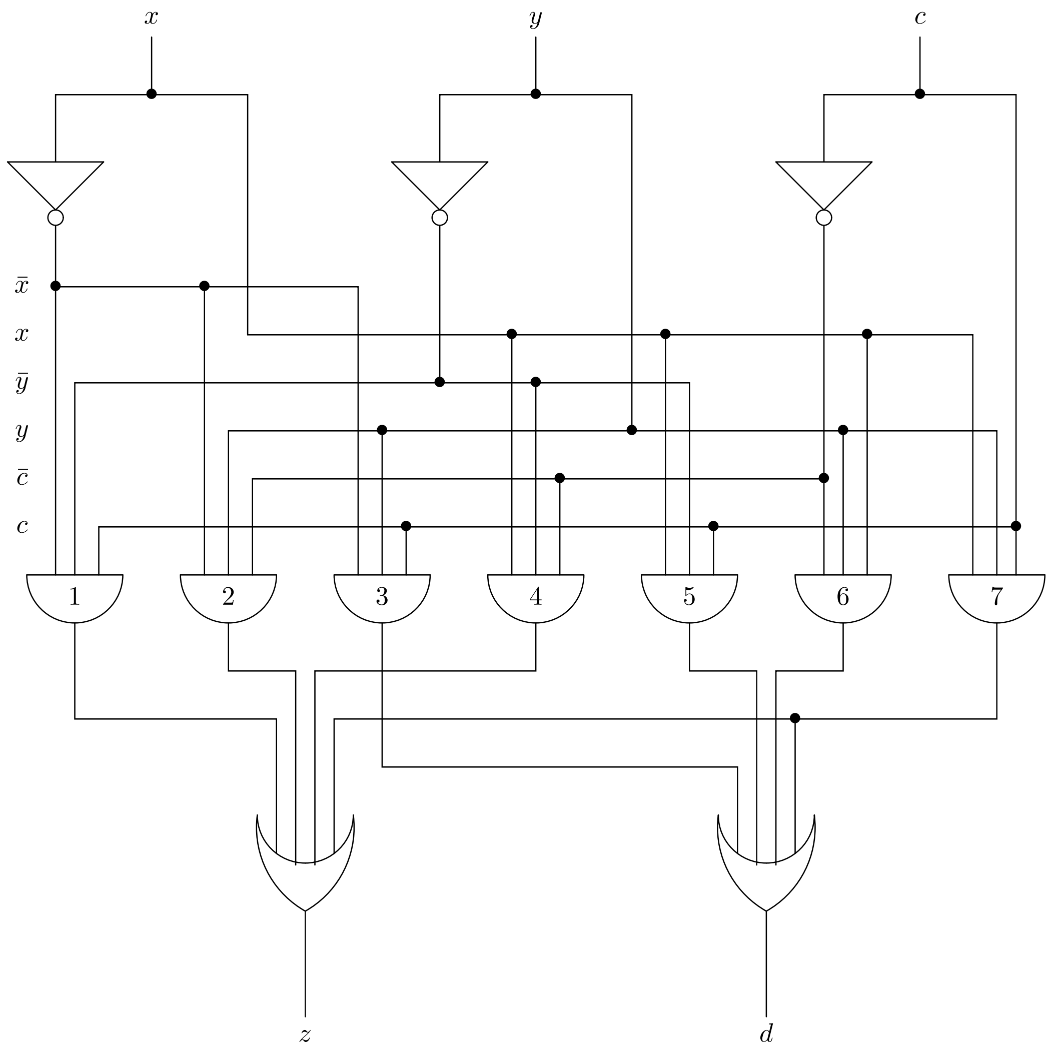


**Fig. 13.10.** One-bit-adder circuit.

binary representation of the number of the gate. Thus, gate 1 has output expression $\bar{x}\bar{y}c$. Finally, the output of the `OR`-gate $z$ is the `OR` of the output expressions for gates 1, 2, 4, and 7, that is

$$\bar{x}\bar{y}c + \bar{x}y\bar{c} + x\bar{y}\bar{c} + xyc$$

Similarly, the output of the `OR`-gate for $d$ is the `OR` of the output expressions for gates 3, 5, 6, and 7, which is

$$\bar{x}yc + x\bar{y}c + xy\bar{c} + xyc$$

We leave it as an exercise to show that this expression is equivalent to the expression

$$yc + xc + xy$$

that we would get if we worked from the Karnaugh map for $d$ alone. ✦

## EXERCISES

**13.4.1**: Design circuits for the following Boolean functions. You need not restrict yourself to 2-input gates if you can group three or more operands that are connected by the same operator.

a) $x + y + z$. *Hint*: Think of this expression as `OR`$(x, y, z)$.
b) $xy + xz + yz$
c) $x + (\bar{y}\bar{x})(y + z)$

**13.4.2**: For each of the circuits in Fig. 13.11, compute the logical expression for each gate. What are the expressions for the outputs of the circuits? For circuit (b) construct an equivalent circuit using only `AND`, `OR`, and `NOT` gates.

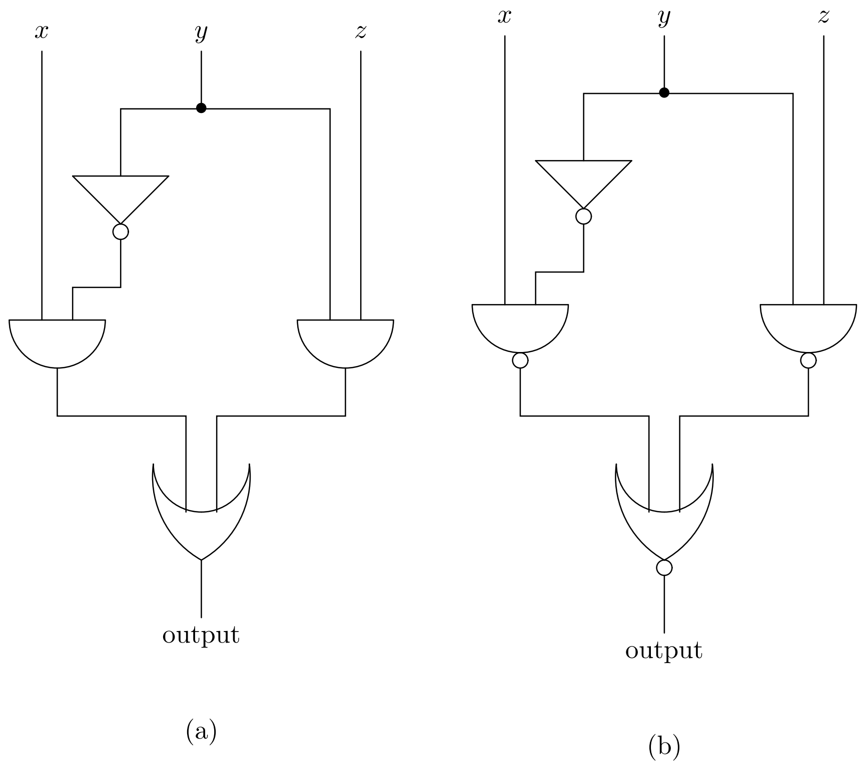


**Fig. 13.11.** Circuits for Exercise 13.4.2.

**13.4.3**: Prove the following tautologies used in Examples 13.4 and 13.5:

a) $(xy + \bar{x}\bar{y}) \equiv (x \equiv y)$
b) $(\bar{x}yc + x\bar{y}c + xy\bar{c} + xyc) \equiv (yc + xc + xy)$

### Chips

Chips generally have several "layers" of material that can be used, in combination, to build gates. Wires can run in any layer, to interconnect the gates; wires on different layers usually can cross without interacting. The "feature size," roughly the minimum width of a wire, is in 1994 usually below half a micron (a *micron* is 0.001 millimeter, or about 0.00004 inches). Gates can be built in an area several microns on a side.

**Feature size**

**Micron**

The process by which chips are fabricated is complex. For example, one step might deposit a thin layer of a certain substance, called a *photoresist*, all over a chip. Then a photographic negative of the features desired on a certain layer is used. By shining light or a beam of electrons through the negative, the top layer can be etched away in places where the beam shines through, leaving only the desired circuit pieces.

## 13.5 Some Physical Constraints on Circuits

**Integrated circuits**

Today, most circuits are built as "chips," or integrated circuits. Large numbers of gates, perhaps as many as millions of gates, and the wires interconnecting them, are constructed out of semiconductor and metallic materials in an area about a centimeter (0.4 inches) on a side. The various "technologies," or methods of constructing integrated circuits, impose a number of constraints on the way efficient circuits can be designed. For example, we mentioned earlier that certain types of gates, such as AND, OR, and NOT, are easier to construct than other kinds.

### Circuit Speed

Associated with each gate is a delay, between the time that the inputs become active and the time that the output becomes available. This delay might be only a few nanoseconds (a nanosecond is $10^{-9}$ seconds), but in a complex circuit, such as the central processing unit of a computer, information propagates through many levels of gates, even during the execution of a single instruction. As modern computers perform instructions in much less than a microsecond (which is $10^{-6}$ seconds), it is evidently imperative that the number of gates through which a value must propagate be kept to a minimum.

Thus, for a combinational circuit, the maximum number of gates that lie along any path from an input to an output is analogous to the running time of a program as a figure of merit. That is, if we want our circuits to compute their outputs fast, we must minimize the longest path length in the graph of the circuit. The *delay* of a circuit is the number of gates on the longest path — that is, one plus the length of the path equals the delay. For example, the adder of Fig. 13.10 has delay 3, since the longest paths from input to output go through one of the inverters, then one of the AND-gates, and finally, through one of the OR-gates; there are many paths of length 3.

**Circuit delay**

Notice that, like running time, circuit delay only makes sense as an "order of magnitude" quantity. Different technologies will give us different values of the time that it takes an input of one gate to affect the output of that gate. Thus, if we have two circuits, of delay 10 and 20, respectively, we know that if implemented in the same technology, with all other factors being equal, the first will take half the time of the second. However, if we implement the second circuit in a faster technology, it could beat the first circuit implemented in the original technology.

## Size Limitations

The cost of building a circuit is roughly proportional to the number of gates in the circuit, and so we would like to reduce the number of gates. Moreover, the size of a circuit also influences its speed, and small circuits tend to run faster. In general, the more gates a circuit has, the greater the area on a chip that it will consume. There are at least two negative effects of using a large area.

**Propagation delay**

1. If the area is large, long wires are needed to connect gates that are located far apart. The longer a wire is, the longer it takes a signal to travel from one end to the other. This *propagation delay* is another source of delay in the circuit, in addition to the time it takes a gate to "compute" its output.

2. There is a limit to how large chips can be, because the larger they are, the more likely it is that there will be an imperfection that causes the chip to fail. If we have to divide a circuit across several chips, then wires connecting the chips will introduce a severe propagation delay.

Our conclusion is that there is a significant benefit to keeping the number of gates in a circuit low.

## Fan-In and Fan-Out Limitations

A third constraint on the design of circuits comes from physical realities. We pay a penalty for gates that have too many inputs or that have their outputs connected to too many other inputs. The number of inputs of a gate is called its *fan-in,* and the number of inputs to which the output of a gate is connected is that gate's *fan-out.* While, in principle, there is no limit on fan-in or fan-out, in practice, gates with large fan-in and/or fan-out will be slower than gates with smaller fan-in and fan-out. Thus, we shall try to design our circuits with limited fan-in and fan-out.

✦ **Example 13.6.** Suppose a particular computer has registers of 32 bits, and we wish to implement, in circuitry, the `COMPARE` machine instruction. One of the things we have to build is a circuit that tests whether a register has all 0's. This test is implemented by an `OR`-gate with 32 inputs, one for each bit of the register. An output of 1 means the register does not hold 0, while an output of 0 means that it does.[1] If we want 1 to mean a positive answer to the question, "Does the register hold 0," then we would complement the output with an inverter, or use a `NOR` gate.

However, a fan-in of 32 is generally much higher than we would like. Suppose we were to limit ourselves to gates with a fan-in of 2. That is probably too low a limit, but will serve for an example. First, how many two-input `OR`-gates do we need to compute the `OR` of $n$ inputs? Clearly, each 2-input gate combines two values into one (its output), and thus reduces by one the number of values we need to compute the `OR` of $n$ inputs. After we have used $n-1$ gates, we shall be down to

[1] Strictly speaking, this observation is true only in 2's complement notation. In some other notations, there are two ways to represent 0. For example, in sign-and-magnitude, we would test only whether the last 31 bits are 0.

one value, and if we have designed the circuit properly, that one value will be the OR of all $n$ original values. Thus, we need at least 31 gates to compute the OR of 32 bits, $x_1, x_2, \ldots, x_{32}$.

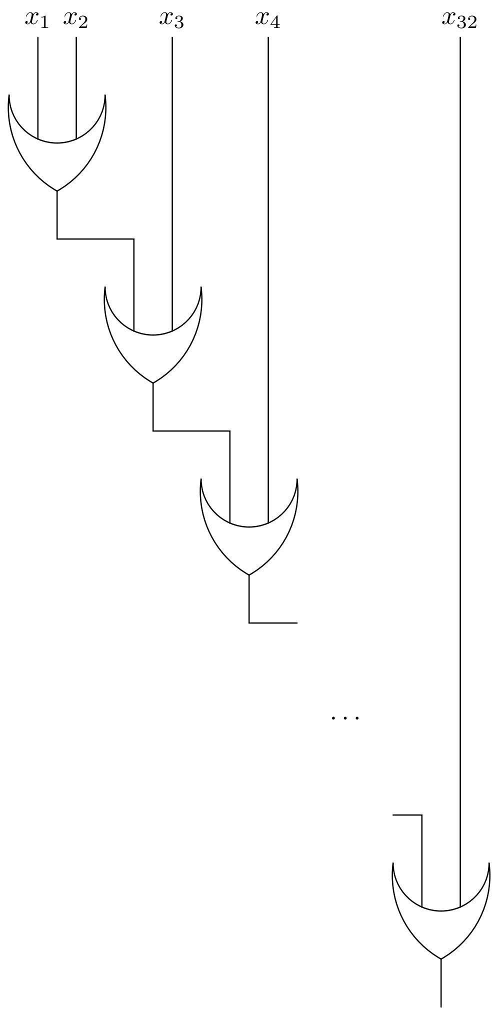


**Fig. 13.12.** Slow way to take the OR of 32 bits.

A naive way to do this OR is shown in Fig. 13.12. There, we group the bits in a left-associative way. As each gate feeds the next, the graph of the circuit has a path with 31 gates, and the delay of the circuit is 31.

A better way is suggested in Fig. 13.13. A complete binary tree with five levels uses the same 31 gates, but the delay is only 5. We would expect the circuit of Fig. 13.13 therefore to run about six times faster than the circuit of Fig. 13.12. Other factors that influence speed might reduce the factor of six, but even for a "small" number of bits like 32, the clever design is significantly faster than the naive design.

If one doesn't immediately "see" the trick of using a complete binary tree as a circuit, one can obtain the circuit of Fig. 13.13 by applying the divide-and-conquer

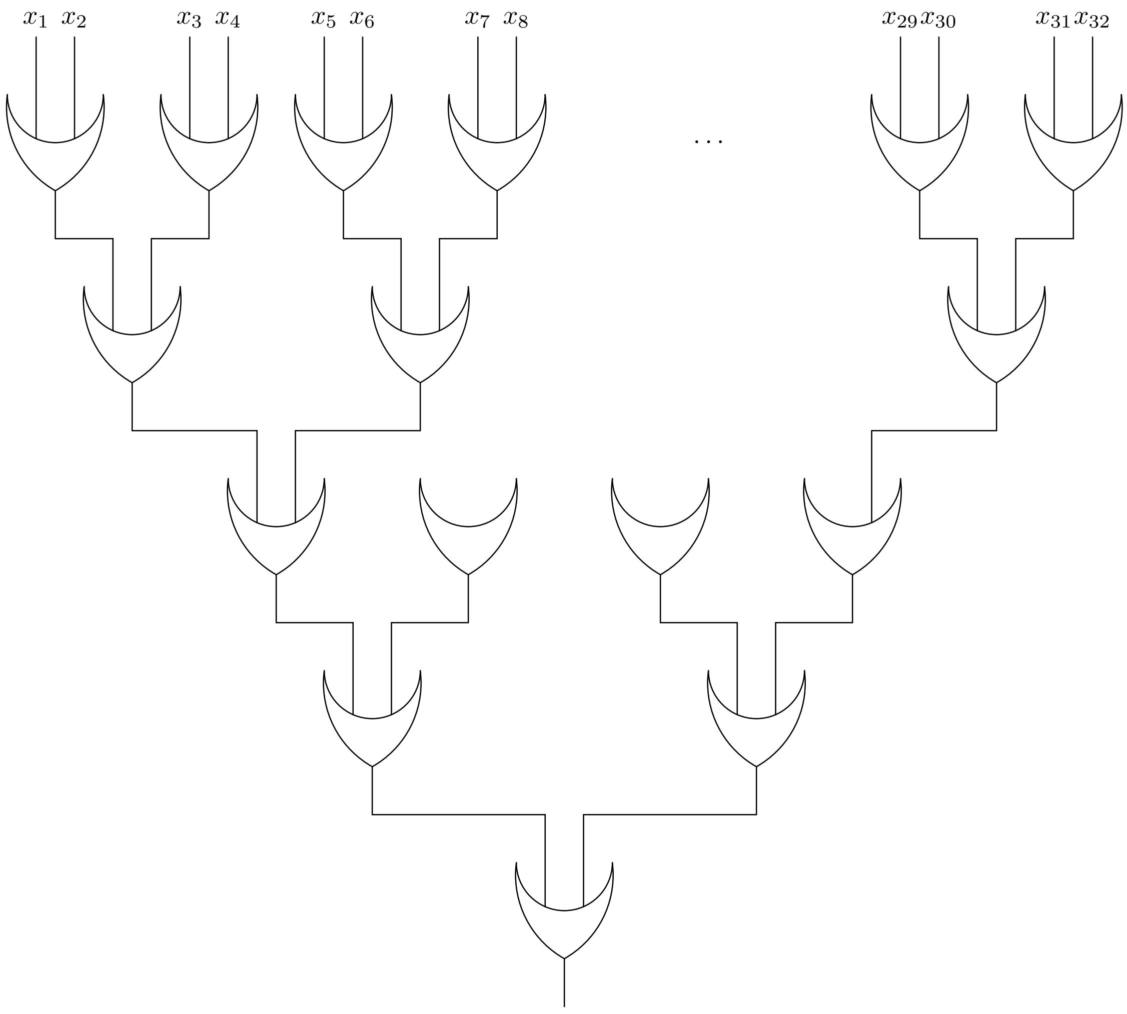


**Fig. 13.13.** Complete binary tree of OR-gates.

**Divide and conquer circuits**

paradigm. That is, to take the OR of $2^k$ bits, we divide the bits into two groups of $2^{k-1}$ bits each. Circuits for each of the two groups are combined by a final OR-gate, as suggested in Fig. 13.14. Of course, the circuit for the basis case $k = 1$ (i.e., two inputs) is provided not by divide-and-conquer, but by using a single two-input OR-gate. ✥

## EXERCISES

**13.5.1***: Suppose that we can use OR-gates with fan-in of $k$, and we wish to take the OR of $n$ inputs, where $n$ is a power of $k$. What is the minimum possible delay for such a circuit? What would be the delay if we used a naive "cascading" circuit as shown in Fig. 13.12?

**13.5.2***: Design divide-and-conquer circuits to perform the following operations. What is the delay of each of your circuits?

a) Given inputs $x_1, x_2, \ldots, x_n$, produce a 1 output if and only if all inputs are 1.

b) Given inputs $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_n$, produce a 1 output if and only if each $x_i$ equals $y_i$, for $i = 1, 2, \ldots, n$. *Hint*: Use the circuit of Fig. 13.2 to test whether two inputs are equal.

**13.5.3***: The divide-and-conquer approach of Fig. 13.14 works even when the number of inputs is not a power of two. Then the basis must include sets of two or three inputs; three-input sets are handled by two `OR`-gates, one feeding the other, assuming we wish to keep strictly to our fan-in limitation of two. What is the delay of such circuits, as a function of the number of inputs?

**13.5.4**: First-string commandos are ready, willing, and able. Suppose we have $n$ commandos, and circuit inputs $r_i$, $w_i$, and $a_i$ indicate, respectively, whether the $i$th commando, is ready, willing, and able. We only want to send the commando team on a raid if they all are ready, willing, and able. Design a divide-and-conquer circuit to indicate whether we can send the team on a raid.

**13.5.5***: Second-string commandos (read Exercise 13.5.4) aren't as professional. We are willing to send them on a raid if each is either ready, willing, or able. In fact, we'll send the team even if at most one of the commandos is neither ready, willing, nor able. Using the same inputs as Exercise 13.5.4, devise a divide-and-conquer circuit that will indicate whether we can send the second-string commando team on a raid.

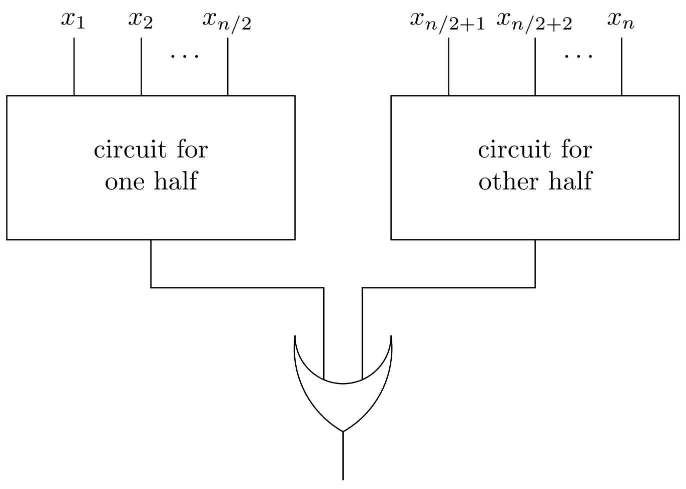


**Fig. 13.14.** Divide-and-conquer approach to circuit design.

## 13.6 A Divide-and-Conquer Addition Circuit

One of the key parts of a computer is a circuit that adds two numbers. While actual microprocessor circuits do more, we shall study the essence of the problem by designing a circuit to add two nonnegative integers. This problem is quite instructive as an example of divide-and-conquer circuit design.

We can build an adder for $n$-bit numbers from $n$ one-bit adders, connected in one of several ways. Let us suppose that we use the circuit of Fig. 13.10 as a one-bit-adder circuit. This circuit has a delay of 3, which is close to the best we can do.[2] The simplest approach to building an adder circuit is the *ripple-carry adder* which we saw in Section 1.3. In this circuit, an output of each one-bit adder becomes an input of the next one-bit adder, so that adding two $n$-bit numbers incurs a delay of $3n$. For example, in the case where $n = 32$, the circuit delay is 96.

### A Recursive Addition Circuit

We can design an adder circuit with significantly less delay if we use the divide-and-conquer strategy of designing a circuit for $n/2$ bits and using two of them, together with some additional circuitry, to make an $n$-bit adder. In Example 13.6, we spoke of a divide-and-conquer circuit for taking the `OR` of many bits, using 2-input `OR`-gates. That was a particularly simple example of the divide-and-conquer technique, since each of the smaller circuits performed exactly the desired function (`OR`), and the combination of outputs of the subcircuits was very simple (they were fed to an `OR`-gate). The two half-size circuits did their work at the same time (in parallel), so their delays did not add.

For the adder, we need to do something more subtle. A naive way to start is to add the left half of the bits (high-order bits) and add the right half of the bits (low-order bits), using identical half-size adder circuits. However, unlike the $n$-bit `OR` example, where we could work on the left and right halves independently, it seems that for the adder, the addition for the left half cannot begin until the right half is finished and passes its carry to the rightmost bit in the left half, as suggested in Fig. 13.15. If so, we shall find that the "divide-and-conquer" circuit is actually identical to the ripple-carry adder, and we have not improved the delay at all.

The additional "trick" we need is to realize that we can begin the computation of the left half without knowing the carry out of the right half, provided we compute more than just the sum. We need to answer two questions. First, what would the sum be if there is no carry into the rightmost place in the left half, and second, what would the sum be if there is a carry-in?[3] We can then allow the circuits for the left and right halves to compute their two answers at the same time. Once both have been completed, we can tell whether or not there is a carry from the right half to the left. That tells us which answer is correct, and with three more levels of delay, we can select the correct answer for the left side. Thus, the delay to add $n$ bits will be just three more than the delay to add $n/2$ bits, leading to a circuit of delay $3(1 + \log_2 n)$. That compares very well with the ripple-carry adder for $n = 32$; the divide-and-conquer adder will have delay $3(1 + \log_2 32) = 3(1 + 5) = 18$, compared with 96 for the ripple-carry adder.

[2] We can design a more complicated one-bit-adder circuit with delay 2 by complementing all the inputs outside the full adder and computing both the carry and its complement within the full adder.

[3] Note "there is a carry-in" means the carry-in is 1; "no carry-in" means the carry-in is 0.

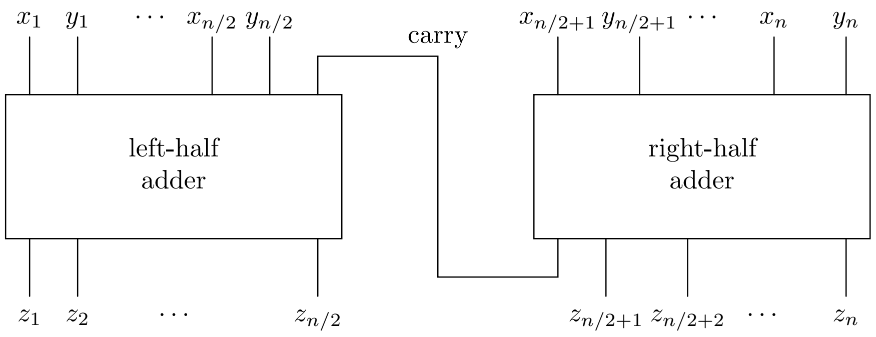

$x_1$ $y_1$ $\cdots$ $x_{n/2}$ $y_{n/2}$
carry
$x_{n/2+1}$ $y_{n/2+1}$ $\cdots$ $x_n$ $y_n$
left-half adder
right-half adder
$z_1$ $z_2$ $\cdots$ $z_{n/2}$
$z_{n/2+1}$ $z_{n/2+2}$ $\cdots$ $z_n$

**Fig. 13.15.** An inefficient divide-and-conquer design for an adder.

**n-adder**

More precisely, we define an *$n$-adder* to be a circuit with inputs $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_n$, representing two $n$-bit integers, and outputs

1. $s_1, s_2, \ldots, s_n$, the $n$-bit sum (excluding a carry out of the leftmost place, i.e., out of the place belonging to $x_1$ and $y_1$) of the inputs, assuming that there is no carry into the rightmost place (the place of $x_n$ and $y_n$).

2. $t_1, t_2, \ldots, t_n$, the $n$-bit sum of the inputs, assuming that there is a carry into the rightmost place.

3. $p$, the *carry-propagate bit,* which is 1 if there is a carry out of the leftmost place, on the assumption that there is a carry into the rightmost place.

4. $g$, the *carry-generate bit,* which is 1 if there is a carry out of the leftmost place, even if there is no carry into the rightmost place.

Note that $g \rightarrow p$; that is, if $g$ is 1, then $p$ must be 1. However, $g$ can be 0, and $p$ still be 1. For example, if the $x$'s are $1010\cdots$, and the $y$'s are $0101\cdots$, then $g = 0$, because when there is no carry in, the sum is all 1's and there is no carry out of the leftmost place. On the other hand, if there is a carry into the rightmost position, then the last $n$ bits of the sum are all 0's, and there is a carry out of the leftmost place; thus $p = 1$.

We shall construct an $n$-adder recursively, for $n$ a power of 2.

**BASIS.** Consider the case $n = 1$. Here we have two inputs, $x$ and $y$, and we need to compute four outputs, $s$, $t$, $p$, and $g$, given by the logical expressions

$$\begin{aligned}
&s = x\bar{y} + \bar{x}y \\
&t = xy + \bar{x}\bar{y} \\
&g = xy \\
&p = x + y
\end{aligned}$$

To see why these expressions are correct, first assume there is no carry into the one place in question. Then the sum bit, which is 1 if an odd number of $x$, $y$, and the carry-in are 1, will be 1 if exactly one of $x$ and $y$ is 1. The expression for $s$ above clearly has that property. Further, with no carry-in, there can only be a

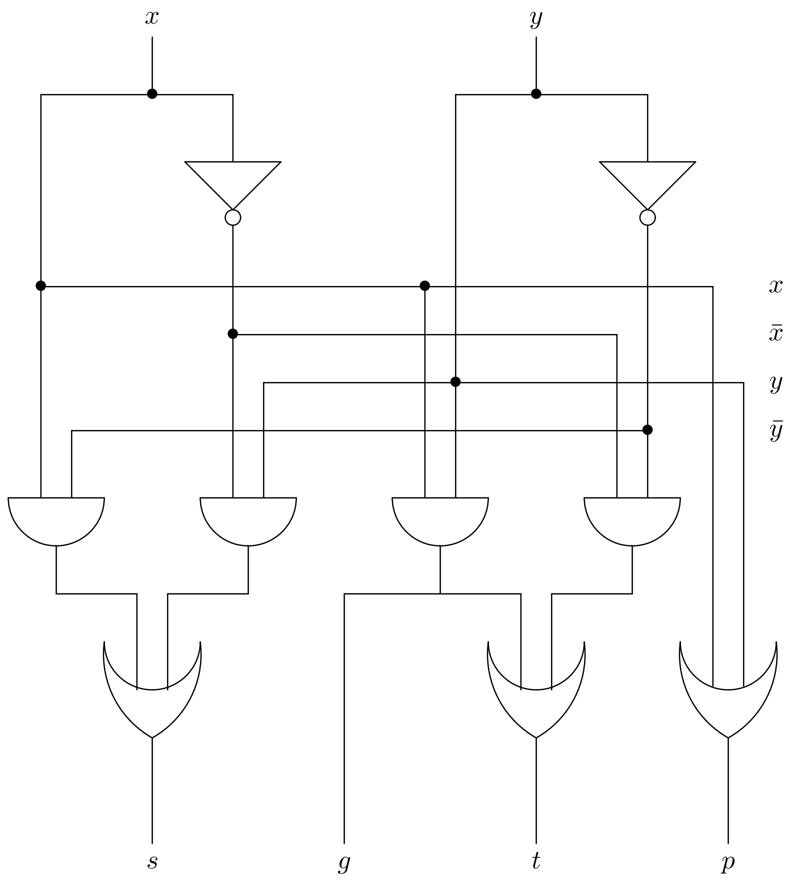


**Fig. 13.16.** Base case: a 1-adder.

carry-out if both $x$ and $y$ are 1, which explains the expression for $g$ above.

Now suppose that there is a carry-in. Then for an odd number of $x$, $y$, and the carry-in to be 1, it must be that both or neither of $x$ and $y$ are 1, explaining the expression for $t$. Also, there will now be a carry-out if either one or both of $x$ and $y$ are 1, which justifies the expression for $p$. A circuit for the basis is shown in Fig. 13.16. It is similar in spirit to the full adder of Fig. 13.10, but is actually somewhat simpler, because it has only two inputs instead of three.

**INDUCTION.** The inductive step is illustrated in Fig. 13.17, where we build a $2n$-adder from two $n$-adders. A $2n$-adder is composed of two $n$-adders, followed by two pieces of circuitry labeled `FIX` in Fig. 13.17, to handle two issues:

1. Computing the carry propagate and generate bits for the $2n$-adder
2. Adjusting the left half of the $s$'s and $t$'s to take into account whether or not there is a carry into the left half from the right

First, suppose that there is a carry into the right end of the entire circuit for the

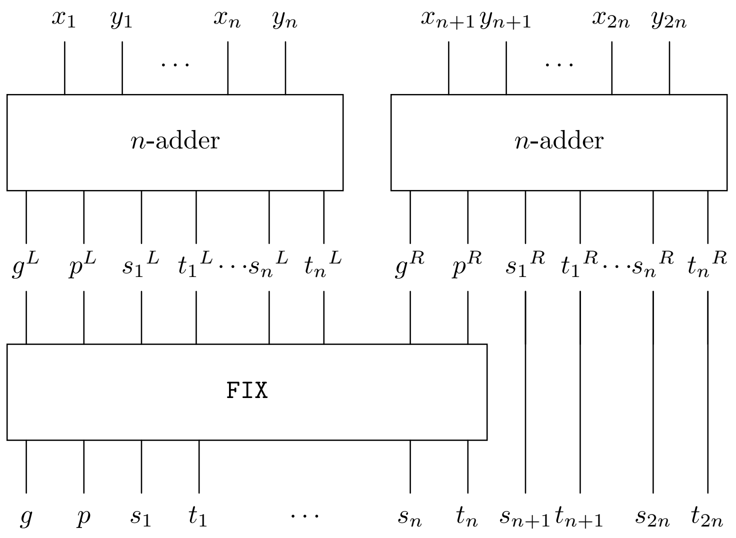

$x_1$ $y_1$ $\cdots$ $x_n$ $y_n$
$x_{n+1}$ $y_{n+1}$ $\cdots$ $x_{2n}$ $y_{2n}$
$n$-adder
$n$-adder
$g^L$ $p^L$ ${s_1}^L$ ${t_1}^L$ $\cdots$ ${s_n}^L$ ${t_n}^L$
$g^R$ $p^R$ ${s_1}^R$ ${t_1}^R$ $\cdots$ ${s_n}^R$ ${t_n}^R$
FIX
$g$ $p$ $s_1$ $t_1$ $\cdots$ $s_n$ $t_n$ $s_{n+1}$ $t_{n+1}$ $s_{2n}$ $t_{2n}$

**Fig. 13.17.** Sketch of the divide-and-conquer adder.

$2n$-adder. Then there will be a carry out at the left end of the entire circuit if either of the following hold:

a) Both halves of the adder propagate a carry; that is, $p^Lp^R$ is true. Note this expression includes the case when the right half generates a carry and the left half propagates it. Then $p^Lg^R$ is true, but $g^R \rightarrow p^R$, so $(p^Lp^R + p^Lg^R) \equiv p^Lp^R$.

b) The left half generates a carry; that is, $g^L$ is true. In this case, the existence of a carry-out on the left does not depend on whether or not there is a carry into the right end, or on whether the right half generates a carry.

Thus, the expression for $p$, the carry-propagate bit for the $2n$-adder, is

$$p = g^L + p^Lp^R$$

Now assume there is no carry-in at the right end of the $2n$-adder. Then there is a carry-out at the left end of the $2n$-adder if either

a) The right half generates a carry and the left half propagates it, or

b) The left half generates a carry.

Thus, the logical expression for $g$ is

$$g = g^L + p^Lg^R$$

Now let us turn our attention to the $s_i$'s and the $t_i$'s. First, the right-half bits are unchanged from the outputs of the right $n$-adder, because the presence of the left half has no effect on the right half. Thus, $s_{n+i} = {s_i}^R$, and $t_{n+i} = {t_i}^R$, for $i = 1, 2, \ldots, n$.

The left-half bits must be modified, however, to take into account the ways in which the right half can generate a carry. First, suppose that there is no carry-in at the right end of the $2n$-adder. This is the situation that the $s_i$'s are supposed to tell us about, so that we can develop expressions for the $s_i$'s on the left, that is, $s_1, s_2, \ldots, s_n$. Since there is no carry-in for the right half, there is a carry-in for the left half only if a carry is generated by the right half. Thus, if $g^R$ is true, then $s_i = {t_i}^L$ (since the ${t_i}^L$'s tell us about what happens when there is a carry into the left half). If $g^R$ is false, then $s_i = {s_i}^L$ (since the ${s_i}^L$'s tell us what happens when there is no carry into the left half). As a logical expression, we can write

$$s_i = {s_i}^L \bar{g}^R + {t_i}^L g^R$$

for $i = 1, 2, \ldots, n$.

Finally, consider what happens when there is a carry-in at the right end of the $2n$-adder. Now we can address the question of the values for the $t_i$'s on the left as follows. There will be a carry into the left half if the right half propagates a carry, that is, if $p^R = 1$. Thus, $t_i$ takes its value from ${t_i}^L$ if $p^R$ is true and from ${s_i}^L$ if $p^R$ is false. As a logical expression,

$$t_i = {s_i}^L \bar{p}^R + {t_i}^L p^R$$

In summary, the circuits represented by the box labeled `FIX` in Fig. 13.17 compute the following expressions:

$$\begin{aligned}
&p = g^L + p^L p^R \\
&g = g^L + p^L g^R \\
&s_i = {s_i}^L \bar{g}^R + {t_i}^L g^R, \text{ for } i = 1, 2, \ldots, n \\
&t_i = {s_i}^L \bar{p}^R + {t_i}^L p^R, \text{ for } i = 1, 2, \ldots, n
\end{aligned}$$

These expressions can each be realized by a circuit of at most three levels. For example, the last expression needs only the circuit of Fig. 13.18.

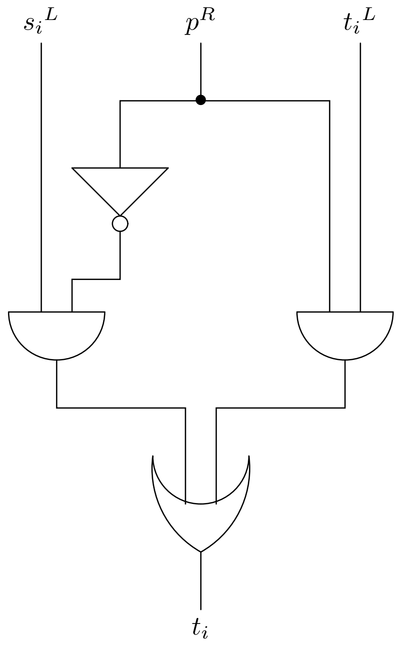


**Fig. 13.18.** Part of the `FIX` circuitry.

## Delay of the Divide-and-Conquer Adder

Let $D(n)$ be the delay of the $n$-adder we just designed. We can write a recurrence relation for $D$ as follows. For the basis, $n = 1$, examine the basis circuit in Fig. 13.16 and conclude that the delay is 3. Thus, $D(1) = 3$.

Now examine the inductive construction of the circuit in Fig. 13.17. The delay of the circuit is the delay of the $n$-adders plus the delay of the `FIX` circuitry. The $n$-adders have delay $D(n)$. Each of the expressions developed for the `FIX` circuitry yields a simple circuit with at most three levels. Figure 13.18 is a typical example. Thus, $D(2n)$ is three more than $D(n)$. The recurrence relation for $D(n)$ is thus

$$\begin{aligned}&D(1) = 3\\&D(2n) = D(n) + 3\end{aligned}$$

The solution, for numbers of bits that are powers of 2, begins $D(1) = 3$, $D(2) = 6$, $D(4) = 9$, $D(8) = 12$, $D(16) = 15$, $D(32) = 18$, and so on. The solution to the recurrence is

$$D(n) = 3(1 + \log_2 n)$$

for $n$ a power of 2, as the reader may check using the methods of Section 3.11. In particular, note that for a 32-bit adder, the delay of 18 is much less than the delay of 96 for the 32-bit ripple-carry adder.

## Number of Gates Used by the Divide-and-Conquer Adder

We should also check that the number of gates is reasonable. Let $G(n)$ be the number of gates used in an $n$-adder circuit. The basis is $G(1) = 9$, as we may see by counting the gates in the circuit of Fig. 13.16. Then we observe that the circuit of Fig. 13.17, the inductive case, has $2G(n)$ gates in the two $n$-adder subcircuits. To this amount, we must add the number of gates in the `FIX` circuitry. As we may invert $g^R$ and $p^R$ once, each of the $n$ $s_i$'s and $t_i$'s can be computed with three gates each (two `AND`'s and an `OR`), or $6n$ gates total. To this quantity we add the two inverters for $g^R$ and $p^R$, and we must add the two gates each that we need to compute $g$ and $p$. The total number of gates in the `FIX` circuitry is thus $6n + 6$. The recurrence for $G$ is hence

$$\begin{aligned}&G(1) = 9\\&G(2n) = 2G(n) + 6n + 6\end{aligned}$$

Again, our function is defined only when $n$ is a power of 2. The first six values of $G$ are tabulated in Fig. 13.19. For $n = 32$, we see that 954 gates are required. The closed-form expression for $G(n)$ is $3n \log_2 n + 15n - 6$, for $n$ a power of 2, as the reader may show by applying the techniques of Section 3.11.

Actually, we can do with somewhat fewer gates, if all we want is a 32-bit adder. For then, we know that there is no carry-in at the right of the 32nd bit, and so the value of $p$, and the values of $t_1, t_2, \ldots, t_{32}$ need not be computed at the last stage of the circuit. Similarly, the right-half 16-adder does not need to compute its carry-propagate bit or its 16 $t$-values; the right-half 8-adder in the right 16-adder does not need to compute its $p$ or $t$'s and so on.

It is interesting to compare the number of gates used by the divide-and-conquer adder with the number of gates used by the ripple-carry adder. The circuit for a full adder that we designed in Fig. 13.10 uses 12 gates. Thus, an $n$-bit ripple-carry

| $n$ | $G(n)$ |
|---|---|
| 1 | 9 |
| 2 | 30 |
| 4 | 78 |
| 8 | 186 |
| 16 | 426 |
| 32 | 954 |

**Fig. 13.19.** Numbers of gates used by various $n$-adders.

adder uses $12n$ gates, and for $n = 32$, this number is 384 (we can save a few gates if we remember that the carry into the rightmost bit is 0).

We see that for the interesting case, $n = 32$, the ripple-carry adder, while much slower, does use fewer than half as many gates as the divide-and-conquer adder. Moreover, the latter's growth rate, $O(n \log n)$, is higher than the growth rate of the ripple-carry adder, $O(n)$, so that the difference in the number of gates gets larger as $n$ grows. However, the ratio is only $O(\log n)$, so that the difference in the number of gates used is not severe. As the difference in the time required by the two classes of circuits is much more significant $[O(n)$ vs. $O(\log n)]$, some sort of divide-and-conquer adder is used in essentially all modern computers.

## EXERCISES

**13.6.1**: Draw the divide-and-conquer circuit, as developed in this section, to add 4-bit numbers.

**13.6.2**: Design circuits similar to Fig. 13.18 to compute the other outputs of the adder in Fig. 13.17, that is, $p$, $g$, and the $s_i$'s.

**13.6.3****: Design a circuit that takes as input a decimal number, with each digit represented by four inputs that give the binary equivalent of that decimal digit. The output is the equivalent number represented in binary. You may assume that the number of digits is a power of 2 and use a divide-and-conquer approach. *Hint*: What information does the left half circuit (high-order digits) need from the right half (low-order digits)?

**13.6.4***: Show that the solution to the recurrence equation

$$D(1) = 3$$
$$D(2n) = D(n) + 3$$

is $D(n) = 3(1 + \log_2 n)$ for $n$ a power of 2.

**13.6.5***: Show that the solution to the recurrence equation

$$G(1) = 9$$
$$G(2n) = 2G(n) + 6n + 6$$

is $G(n) = 3n \log_2 n + 15n - 6$ for $n$ a power of 2.

**13.6.6****: We observed that if all we want is a 32-bit adder, we do not need all 954 gates as was indicated in Fig. 13.19. The reason is that we can assume there is no carry into the rightmost place of the 32 bits. How many gates do we really need?

## 13.7 Design of a Multiplexer

**Control and data inputs**

A *multiplexer,* often abbreviated MUX, is a common form of computer circuit that takes $d$ *control inputs,* say $x_1, x_2, \ldots, x_d$, and $2^d$ *data inputs,* say $y_0, y_1, \ldots, y_{2^d-1}$, as shown in Fig. 13.20. The output of the MUX is equal to one particular data input, the input $y_{(x_1x_2\cdots x_d)_2}$. That is, we treat the control inputs as a binary integer in the range 0 to $2^d-1$. This integer is the subscript of the data input that we pass to the output.

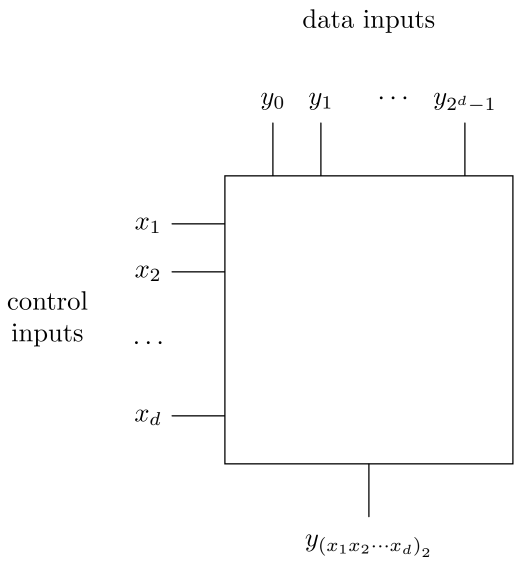
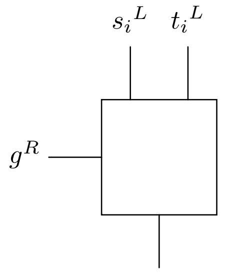


**Fig. 13.20.** A multiplexer circuit schematic.

**Example 13.7.** The circuits computing $s_i$ and $t_i$ in the divide-and-conquer adder are multiplexers with $d=1$. For instance, the formula for $s_i$ is ${s_i}^L\bar{g}^R + {t_i}^L g^R$ and its circuit schematic is shown in Fig. 13.21. Here, $g^R$ plays the role of the control input $x_1$, ${s_i}^L$ is the data input $y_0$, and ${t_i}^L$ is the data input $y_1$.

As another example, the formula for the output of a MUX with two control inputs, $x_1$ and $x_2$, and four data inputs, $y_0$, $y_1$, $y_2$, and $y_3$, is

$$y_0\bar{x_1}\bar{x_2} + y_1\bar{x_1}x_2 + y_2x_1\bar{x_2} + y_3x_1x_2$$



$$s_i = s_i{}^L\bar{g}^R + t_i{}^L g^R$$

**Fig. 13.21.** A 1-multiplexer.

Notice that there is one term for each data input. The term with data input $y_i$ also has each of the control inputs, either negated or unnegated. We can tell which are negated by writing $i$ as a $d$-bit binary integer. If the $j$th position of $i$ in binary has 0, then $x_j$ is negated, and if the $j$th position has 1, we do not negate $x_j$. Note that this rule works for any number $d$ of control inputs. ✦

The straightforward design for a multiplexer is a circuit with three levels of gates. At the first level, we compute the negations of the control bits. The next level is a row of AND-gates. The $i$th gate combines the data input $y_i$ with the appropriate combination of control inputs and negated control inputs. Thus, the output of the $i$th gate is always 0 unless the control bits are set to the binary representation of $i$, in which case the output is equal to $y_i$. The final level is an OR-gate with inputs from each of the AND-gates. As all the AND-gates but one have output 0, the remaining gate, say the $i$th, which has output $y_i$, makes the output of the circuit equal to whatever $y_i$ is. An example of this circuit for $d = 2$ is shown in Fig. 13.22.

## A Divide-and-Conquer Multiplexer

The circuit of Fig. 13.22 has maximum fan-in 4, which is generally acceptable. However, as $d$ gets larger, the fan-in of the OR-gate, which is $2^d$, grows unacceptably. Even the AND-gates, with $d + 1$ inputs each, begin to have uncomfortably large fan-in. Fortunately, there is a divide-and-conquer approach based on splitting the control bits in half, that allows us to build the circuit with gates of fan-in at most 2. Moreover, this circuit uses many fewer gates and is almost as fast as the generalization of Fig. 13.22, provided we require that all circuits be built of gates with the same limit on fan-in.

An inductive construction of a family of multiplexer circuits follows: We call the circuit for a multiplexer with $d$-control-inputs and $2^d$-data-inputs a $d$-MUX.

**BASIS.** The basis is a multiplexer circuit for $d = 1$, that is, a 1-MUX, which we show in Fig. 13.23. It consists of four gates, and the fan-in is limited to 2.

**INDUCTION.** The induction is performed by the circuit in Fig. 13.24, which constructs a $2d$-MUX from $2^d + 1$ copies of $d$-MUX's. Notice that while we double the number of control inputs, we square the number of data inputs, since $2^{2d} = (2^d)^2$.

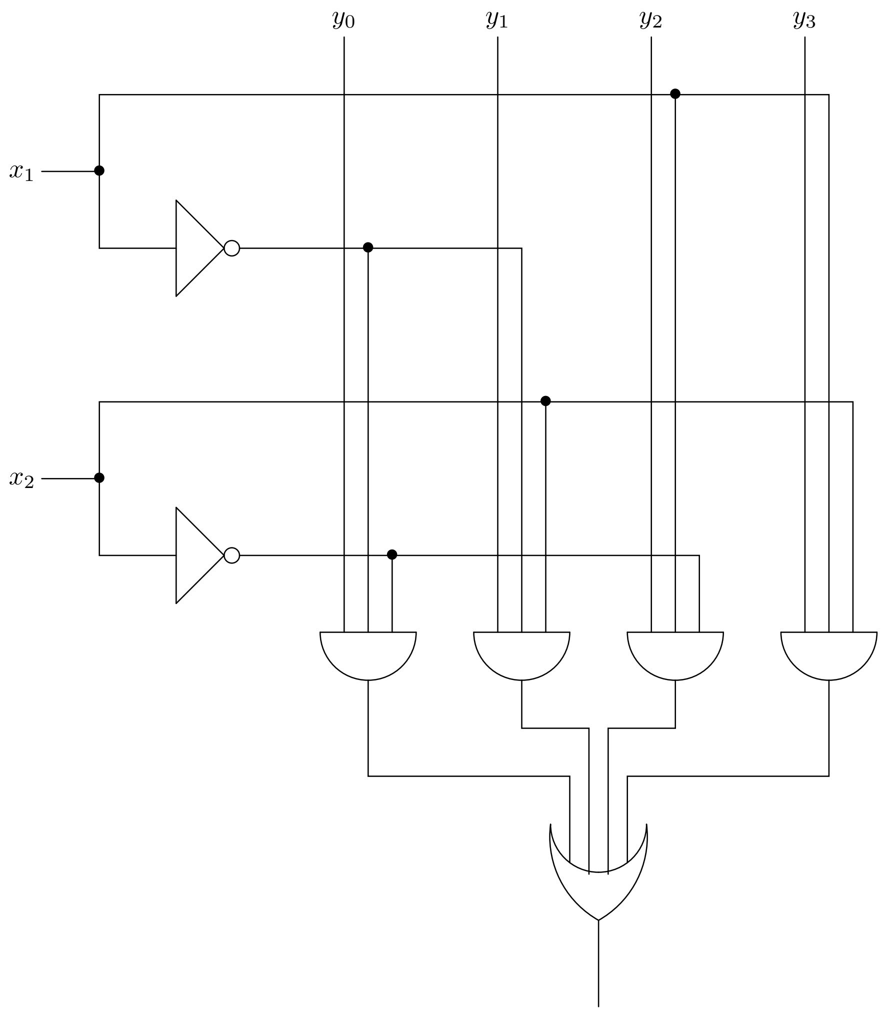


**Fig. 13.22.** Multiplexer circuit for $d = 2$.

Suppose that the control inputs to the $2d$-MUX call for data input $y_i$; that is,

$$i = (x_1 x_2 \cdots x_{2d})_2$$

Each $d$-MUX in the top row of Fig. 13.24 takes a group of $2^d$ data inputs, starting with some $y_j$, where $j$ is a multiple of $2^d$. Thus, if we use the low-order $d$ control bits, $x_{d+1}, \ldots, x_{2d}$, to control each of these $d$-MUX's, the selected input is the $k$th from each group (counting the leftmost in each group as input 0), where

$$k = (x_{d+1} \cdots x_{2d})_2$$

That is, $k$ is the integer represented by the low-order half of the bits.

The data inputs to the bottom $d$-MUX are the outputs of the top row of $d$-MUX's, which we just discovered are $y_k, y_{2^d+k}, y_{2\times 2^d+k}, \ldots, y_{(2^d-1)2^d+k}$. The

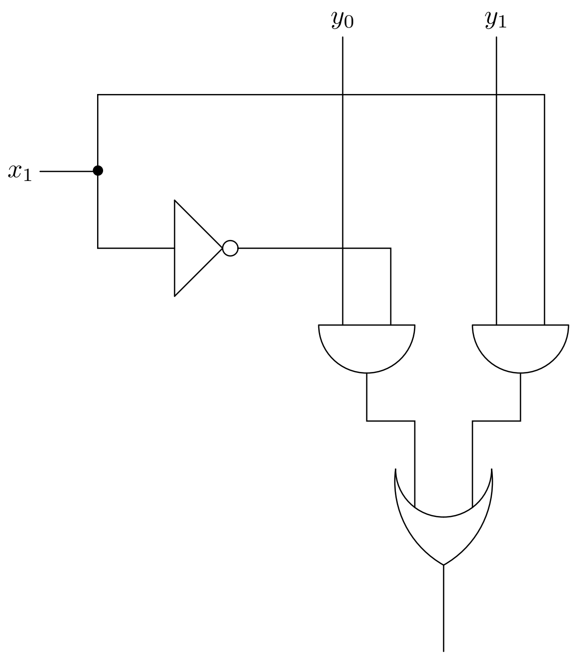


**Fig. 13.23.** Basis circuit, the multiplexer for $d = 1$.

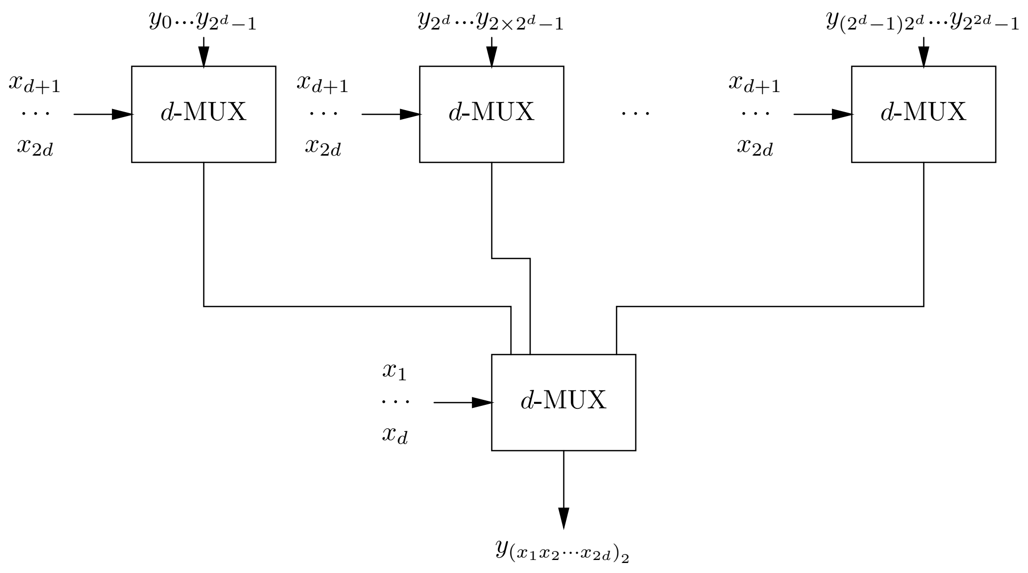

$y_0 \cdots y_{2^d-1}$
$y_{2^d} \cdots y_{2\times 2^d-1}$
$y_{(2^d-1)2^d} \cdots y_{2^{2d}-1}$
$x_{d+1}$ $\cdots$ $x_{2d}$
$d$-MUX
$x_1$ $\cdots$ $x_d$
$y_{(x_1 x_2 \cdots x_{2d})_2}$

**Fig. 13.24.** Divide-and-conquer multiplexer.

bottom $d$-MUX is controlled by $x_1 \cdots x_d$, which represents some integer $j$ in binary; that is, $j = (x_1 \cdots x_d)_2$. The bottom MUX thus selects as its output the $j$th of its inputs (counting the leftmost as input 0). The selected output is thus $y_{j2^d+k}$.

We can show that $j2^d + k = i$ as follows. Notice that multiplying $j$ by $2^d$ has the effect of shifting the binary representation of $j$ left by $d$ places. That is, $j2^d = (x_1 \cdots x_d 0 \cdots 0)_2$, where the string of 0's shown is of length $d$. Thus, the binary representation of $j2^d + k$ is $(x_1 \cdots x_d x_{d+1} \cdots x_{2d})_2$. That follows because the binary representation of $k$ is $(x_{d+1} \cdots x_{2d})_2$, and there is evidently no carry out of the $d$th place from the right, when this number is added to the number $j2^d$, which ends in $d$ 0's. We now see that $j2^d + k = i$, because they have the same binary representations. Thus, the $2d$-MUX of Fig. 13.24 correctly selects $y_i$, where $i = (x_1 \cdots x_{2d})_2$.

## Delay of the Divide-and-Conquer MUX

We can calculate the delay of the multiplexer circuit we designed by writing the appropriate recurrence. Let $D(d)$ be the delay of a $d$-MUX. Inspection of Fig. 13.23 tells us that for $d = 1$, the delay is 3. However, to get a tighter bound, we shall assume that all control inputs are passed through inverters outside the MUX, and not count the level for inverters in Fig. 13.23. We shall then add 1 to the total delay, to account for inversion of all the control inputs, after we determine the delay of the rest of the circuit. Thus, we shall start our recurrence with $D(1) = 2$.

For the induction, we note that the delay through the circuit of Fig. 13.24 is the sum of the delays through any one of the MUX's in the upper row, and the delay through the final MUX. Thus, $D(2d)$ is twice $D(d)$, and we have the recurrence

$$D(1) = 2$$
$$D(2d) = 2D(d)$$

The solution is easy to find. We have $D(2) = 4$, $D(4) = 8$, $D(8) = 16$, and in general, $D(d) = 2d$. Of course, technically this formula only holds when $d$ is a power of 2, but the same idea can be used for an arbitrary number of control bits $d$. Since we must add 1 to the delay for the inversion of the control inputs, the total delay of the circuit is $2d + 1$.

Now consider the simple multiplexer circuit (an `AND` for each data input, all feeding one `OR`-gate). As stated, its delay is 3, independent of $d$, but we cannot generally build it because the fan-in of the final `OR`-gate is unrealistic. What happens if we insist on limiting the fan-in to 2? Then the `OR`-gate, with $2^d$ inputs, is replaced by a complete binary tree with $d$ levels. Recall that such a tree will have $2^d$ leaves, exactly the right number. The delay of this tree is $d$.

We also have to replace the `AND`-gates with trees of fan-in 2 gates, since in general the `AND`-gates have $d + 1$ inputs. Recall that when using gates with two inputs, each use of a gate reduces the number of inputs by 1, so that it takes $d$ gates of fan-in 2 to reduce $d + 1$ inputs to one output. If we arrange the gates as a balanced binary tree of `AND`-gates, we need $\lceil \log_2 d{+}1 \rceil$ levels. When we add one more level for inverting the control inputs, we have a total delay of $d + 1 + \lceil \log_2(d + 1) \rceil$. This figure compares favorably with the delay $2d + 1$ for the divide-and-conquer MUX, although the difference is not great, as shown in the table of Fig. 13.25.

| | Delay | |
|---|---|---|
| $d$ | Divide-and-conquer MUX | Simple MUX |
| 1 | 3 | 3 |
| 2 | 5 | 5 |
| 4 | 9 | 8 |
| 8 | 17 | 13 |
| 16 | 33 | 22 |

**Fig. 13.25.** Delay of multiplexer designs.

## Gate Count

In this section we compare the number of gates between the simple MUX and the divide-and-conquer MUX. We shall see that the divide-and-conquer MUX has strikingly fewer gates as $d$ increases.

To count the number of gates in the divide-and-conquer MUX, we can temporarily ignore the inverters. We know that each of the $d$ control inputs is inverted once, so that we can just add $d$ to the count at the end. Let $G(d)$ be the number of gates (excluding inverters) used in the $d$-MUX. Then we can develop a recurrence for $G$ as follows:

**BASIS.** For the basis case, $d = 1$, there are three gates in the circuit of Fig. 13.23, excluding the inverter. Thus, $G(1) = 3$.

**INDUCTION.** For the induction, the $2d$-MUX in Fig. 13.24 is built entirely from $2^d + 1$ $d$-MUX's.

Thus, the recurrence relation is

$$G(1) = 3$$

$$G(2d) = (2^d + 1)G(d)$$

As we saw in Section 3.11, the solution to this recurrence is

$$G(d) = 3(2^d - 1)$$

The first few values of the recurrence are $G(2) = 9$, $G(4) = 45$, and $G(8) = 765$.

Now consider the number of gates used in the simple MUX, converted to use only gates of fan-in 2. As before, we shall ignore the $d$ inverters needed for the control inputs. The final `OR`-gate is replaced by a tree of $2^d - 1$ `OR`-gates. Each of the $2^d$ `AND`-gates is replaced by a tree of $d$ `AND`-gates. Thus, the total number of gates is $2^d(d + 1) - 1$. This function is greater than the number of gates for the divide-and-conquer MUX, approximately by the ratio $(d + 1)/3$. Figure 13.26 compares the gate counts (excluding the $d$ inverters in each case) for the two kinds of MUX.

| GATE COUNT | | |
|---|---|---|
| $d$ | Divide-and-conquer MUX | Simple MUX |
| 1 | 3 | 3 |
| 2 | 9 | 11 |
| 4 | 45 | 79 |
| 8 | 765 | 2303 |
| 16 | 196,605 | 1,114,111 |

**Fig. 13.26.** Gate count for multiplexer designs (excludes inverters).

## More About Divide-and-Conquer

The style of divide-and-conquer algorithm represented by our multiplexer design is a rare, but powerful, form. Most examples of divide-and-conquer split a problem into two parts. Examples are merge sort, the fast adder developed in Section 13.6, and the complete binary tree used to compute the AND or OR of a large number of bits. In the multiplexer, we build a $2d$-MUX from $d+1$ smaller MUX's. Put another way, a MUX for $n = 2^{2d}$ data inputs is built from $\sqrt{n}+1$ small MUX's.

## EXERCISES

**13.7.1**: Using the divide-and-conquer technique of this section, construct a

a) 2-MUX
b) 3-MUX

**13.7.2***: How would you construct a multiplexer for which the number of data inputs is not a power of two?

**One-hot decoder**

**13.7.3***: Use the divide-and-conquer technique to design a *one-hot-decoder.* This circuit takes $d$ inputs, $x_1, x_2, \ldots, x_d$ and has $2^d$ outputs $y_0, y_1, \ldots, y_{2^d-1}$. Exactly one of the outputs will be 1, specifically that $y_i$ such that $i = (x_1, x_2, \ldots, x_d)_2$. What is the delay of your circuit as a function of $d$? How many gates does it use as a function of $d$? *Hint*: There are several approaches. One is to design the circuit for $d$ by taking a one-hot-decoder for the first $d-1$ inputs and splitting each output of that decoder into two outputs based on the last input, $x_d$. A second is to assume $d$ is a power of 2 and start with two one-hot-decoders, one for the first $d/2$ inputs and the other for the last $d/2$ inputs. Then combine the outputs of these decoders appropriately.

**13.7.4***: How does your circuit for Exercise 13.7.3 compare, in delay and number of gates, with the obvious one-hot-decoder formed by creating one AND-gate for each output and feeding to that gate the appropriate inputs and inverted inputs? How does the circuit of Exercise 13.7.3 compare with your circuit of this exercise if you replace AND-gates with large fan-in by trees of 2-input gates?

**Majority circuit**

**13.7.5***: A *majority circuit* takes $2d-1$ inputs and has a single output. Its output is 1 if $d$ or more of the inputs are 1. Design a divide-and-conquer majority circuit. What are its delay and gate count as a function of $d$? *Hint*: Like the adder of Section 13.6, this problem is best solved by a circuit that computes more than we need. In particular, we can design a circuit that takes $n$ inputs and has $n+1$ outputs, $y_0, y_1, \ldots, y_n$. Output $y_i$ is 1 if exactly $i$ of the inputs are 1. We can then construct the majority circuit inductively by either of the two approaches suggested in Exercise 13.7.3.

**13.7.6***: There is a naive majority circuit that is constructed by having one `AND` gate for every set of $d$ inputs. The output of the majority circuit is the `OR` of all these `AND`-gates. How do the delay and gate count of the naive circuit compare with that of the divide-and-conquer circuit of Exercise 13.7.5? What if the gates of the naive circuit are replaced by 2-input gates?

## ✦ 13.8 Memory Elements

Before leaving the topic of logic circuits, let us consider a very important type of circuit that is sequential rather than combinational. A *memory element* is a collection of gates that can remember its last input and produce that input at its output, no matter how long ago that input was given. The main memory of the computer consists of bits that can be stored into and that will hold their value until another value is stored.

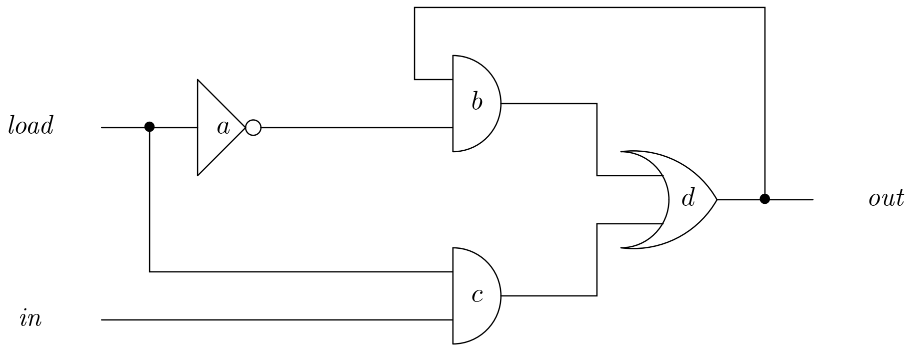


**Fig. 13.27.** A memory element.

Figure 13.27 is a simple memory element. It is controlled by an input called *load*. Ordinarily, *load* has value 0. In that case, the output of inverter $a$ is 1. Since an `AND`-gate has output 0 whenever one or more of its inputs is 0, the output of `AND`-gate $c$ must be 0 whenever *load* is 0.

If $load = 0$ and the output of gate $d$ (which is also the circuit output) is 1, then both inputs to gate $b$ are 1, and so its output is 1. Thus, one of the inputs to `OR`-gate $d$ is 1, and so its output remains 1. On the other hand, suppose the output of $d$ is 0. Then an input to `AND`-gate $b$ is 0, which means that its output is 0. That makes both inputs to $d$ be 0, and so the output remains 0 as long as $load = 0$. We conclude that while $load = 0$, the circuit output remains what it was.

### Real Memory Chips

We should not imagine that Fig. 13.27 represents precisely a typical register bit, but it is not too deceptive. While it also represents a bit of main memory, at least in principle, there are significant differences, and many of the issues in the design of a memory chip involve electronics at a level of detail well beyond the scope of the book.

Because memory chips are used in such great quantities, both in computers and other kinds of hardware, their large-scale production has made feasible some subtle designs for chips storing a million bits or more. To get an idea of the compactness of a memory chip, recall its area is about a square centimeter ($10^{-4}$ square meter). If there are 16 million bits on the chip, then each bit occupies an area equal to $6 \times 10^{-12}$ square meters, or an area about 2.5 microns on a side (remember, a micron is $10^{-6}$ meter). If the minimum width of a wire, or the space between wires, is 0.3 micron, that doesn't leave much room for circuitry to build a memory element. To make matters worse, we need not only to store bits, but also to select one of the 16 million bits to receive a value or one of the 16 million to have its value read. The selection circuitry takes up a significant fraction of the space on the chip, leaving even less space for the memory element itself.

Now consider what happens when $load = 1$. The output of inverter $a$ is now 0, so that the output of AND-gate $b$ will be 0 as well. On the other hand, the first input to AND-gate $c$ is 1, so that the output of $c$ will be whatever the input *in* is. Likewise, as the first input to OR-gate $d$ is 0, the output of $d$ will be the same as the output of $c$, which in turn is the same as circuit input *in*. Thus, setting *load* to 1 causes the circuit output to become whatever *in* is. When we change *load* back to 0, that circuit output continues to circulate between gates $b$ and $d$, as discussed.

We conclude that the circuit of Fig. 13.27 behaves like a memory element, if we interpret "circuit input" as meaning whatever value *in* has at a time when *load* is 1. If *load* is zero, then we say there is no circuit input, regardless of the value of *in*. By setting *load* to 1, we can cause the memory element to accept a new value. The element will hold that value as long as *load* is 0, that is, as long as there is no new circuit input.

### EXERCISES

**13.8.1**: Draw a timing diagram similar to that in Fig. 13.6 for the memory-element circuit shown in Fig. 13.27.

**13.8.2**: Describe what happens to the behavior of the memory element shown in Fig. 13.27 if an alpha particle hits the inverter and for a short time (but enough time for signals to propagate around the circuit) causes the output of gate $a$ to be the same as its input.

## 13.9 Summary of Chapter 13

After reading this chapter, the reader should have more familiarity with the circuitry in a computer and how logic can be used to help design this circuitry. In particular, the following points were covered:

- What gates are and how they are combined to form circuits
- The difference between a combinational circuit and a sequential circuit
- How combinational circuits can be designed from logical expressions, and how logical expressions can be used to model combinational circuits
- How algorithm-design techniques such as divide-and-conquer can be used to design circuits such as adders and multiplexers
- Some of the factors that go into the design of fast circuits
- An indication of how a computer stores bits in its electronic circuitry

## 13.10 Bibliographic Notes for Chapter 13

Shannon [1938] was the first to observe that Boolean algebra can be used to describe the behavior of combinational circuits. For a more comprehensive treatment on the theory and design of combinational circuits, see Friedman and Menon [1975].

Mead and Conway [1980] describe techniques used to construct very large scale integrated circuits. Hennessy and Patterson [1990] discuss computer architecture and the techniques for organizing its circuit elements.

Friedman, A. D., and P. R. Menon [1975]. *Theory and Design of Switching Circuits*, Computer Science Press, New York.

Hennessy, J. L., and D. A. Patterson [1990]. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, Calif.

Mead, C., and L. Conway [1980]. *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass.

Shannon, C. E. [1938]. "Symbolic analysis of relay and switching circuits," *Trans. of AIEE* **57**, pp. 713–723.