28 Arithmetic Circuits

The model of computation provided by an ordinary computer assumes that the basic arithmetic operations—addition, subtraction, multiplication, and division—can be performed in constant time. This abstraction is reasonable, since most basic operations on a random-access machine have similar costs. When it comes to designing the circuitry that implements these operations, however, we soon discover that performance depends on the magnitudes of the numbers being operated on. For example, we all learned in grade school how to add two natural numbers, expressed as n-digit decimal numbers, in $\Theta(n)$ steps (although teachers usually do not emphasize the number of steps required).

This chapter introduces circuits that perform arithmetic functions. With serial processes, $\Theta(n)$ is the best asymptotic time bound we can hope to achieve for adding two n-digit numbers. With circuits that operate in parallel, however, we can do better. In this chapter, we shall design circuits that can quickly perform addition and multiplication. (Subtraction is essentially the same as addition, and division is deferred to Problem 28-1.) We shall assume that all inputs are n-bit natural numbers, expressed in binary.

We start in Section 28.1 by presenting combinational circuits. We shall see how the depth of a circuit corresponds to its "running time." The full adder, which is a building block of most of the circuits in this chapter, serves as our first example of a combinational circuit. Section 28.2 presents two combinational circuits for addition: the ripple-carry adder, which works in $\Theta(n)$ time, and the carry-lookahead adder, which takes only $O(\lg n)$ time. It also presents the carry-save adder, which can reduce the problem of summing three numbers to the problem of summing two numbers in $\Theta(1)$ time. Section 28.3 introduces two combinational multipliers: the array multiplier, which takes $\Theta(n)$ time, and the Wallace-tree multiplier, which requires only $\Theta(\lg n)$ time. Finally, Section 28.4 presents circuits with clocked storage elements (registers) and shows how hardware can be saved by reusing combinational circuitry.

28.1 Combinational circuits

Like the comparison networks of Chapter 27, combinational circuits operate in ***parallel***: many elements can compute values simultaneously as a single step. In this section, we define combinational circuits and investigate how larger combinational circuits can be built up from elementary gates.

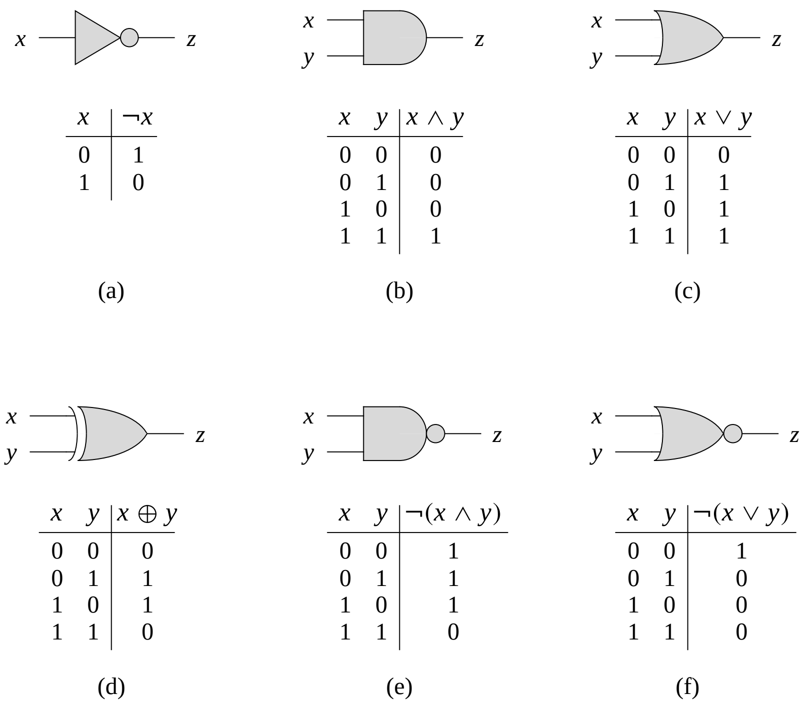
Combinational elements

Arithmetic circuits in real computers are built from combinational elements that are interconnected by wires. A ***combinational element*** is any circuit element that has a constant number of inputs and outputs and that performs a well-defined function. Some of the elements we shall deal with in this chapter are ***boolean combinational elements***—their inputs and outputs are all drawn from the set $\{0, 1\}$, where 0 represents FALSE and 1 represents TRUE.

A boolean combinational element that computes a simple boolean function is called a ***logic gate***. Figure 28.1 shows the four basic logic gates that will serve as combinational elements in this chapter: the ***NOT gate*** (or ***inverter***), the ***AND gate***, the ***OR gate***, and the ***XOR gate***. (It also shows two other logic gates—the ***NAND gate*** and the ***NOR gate***—that are required by some of the exercises.) The NOT gate takes a single binary ***input*** x, whose value is either 0 or 1, and produces a binary ***output*** z whose value is opposite that of the input value. Each of the other three gates takes two binary inputs x and y and produces a single binary output z.

The operation of each gate, and of any boolean combinational element, can be described by a ***truth table***, shown under each gate in Figure 28.1. A truth table gives the outputs of the combinational element for each possible setting of the inputs. For example, the truth table for the XOR gate tells us that when the inputs are $x = 0$ and $y = 1$, the output value is $z = 1$; it computes the "exclusive OR" of its two inputs. We use the symbols $\neg$ to denote the NOT function, $\wedge$ to denote the AND function, $\vee$ to denote the OR function, and $\oplus$ to denote the XOR function. Thus, for example, $0 \oplus 1 = 1$.

Combinational elements in real circuits do not operate instantaneously. Once the input values entering a combinational element ***settle***, or become ***stable***—that is, hold steady for a long enough time—the element's output value is guaranteed to become both stable and correct a fixed amount of time later. We call this time differential the ***propagation delay*** of the element. We assume in this chapter that all combinational elements have constant propagation delay.



x	$\neg x$
0	1
1	0

(a)

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

(b)

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

(c)

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

(d)

x	y	$\neg(x \wedge y)$
0	0	1
0	1	1
1	0	1
1	1	0

(e)

x	y	$\neg(x \vee y)$
0	0	1
0	1	0
1	0	0
1	1	0

(f)

Figure 28.1 Six basic logic gates, with binary inputs and outputs. Under each gate is the truth table that describes the gate's operation. **(a)** The NOT gate. **(b)** The AND gate. **(c)** The OR gate. **(d)** The XOR (exclusive-OR) gate. **(e)** The NAND (NOT-AND) gate. **(f)** The NOR (NOT-OR) gate.

Combinational circuits

A ***combinational circuit*** consists of one or more combinational elements interconnected in an acyclic fashion. The interconnections are called ***wires***. A wire can connect the output of one element to the input of another, thereby providing the output value of the first element as an input value of the second. Although a single wire may have no more than one combinational-element output connected to it, it can feed several element inputs. The number of element inputs fed by a wire is called the ***fan-out*** of the wire. If no element output is connected to a wire, the wire is a ***circuit input***, accepting input values from an external source. If no element input is connected to a wire, the wire is a ***circuit output***, providing the results of the circuit's computation to the outside world. (An internal wire can also fan out to a circuit output.) Combinational circuits contain no cycles and have no memory elements (such as the registers described in Section 28.4).

Full adders

As an example, Figure 28.2 shows a combinational circuit, called a ***full adder***, that takes as input three bits x, y, and z. It outputs two bits, s and c, according to the following truth table:

x	y	z	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Output s is the ***parity*** of the input bits,

$$s = \text{parity}(x, y, z) = x \oplus y \oplus z \ , \tag{28.1}$$

and output c is the ***majority*** of the input bits,

$$c = \text{majority}(x, y, z) = (x \wedge y) \vee (y \wedge z) \vee (x \wedge z) \ . \tag{28.2}$$

(In general, the parity and majority functions can take any number of input bits. The parity is 1 if and only if an odd number of the inputs are 1's. The majority is 1 if and only if more than half the inputs are 1's.) Note that the c and s bits, taken together, give the sum of x, y, and z. For example, if $x = 1$, $y = 0$, and $z = 1$, then $\langle c, s \rangle = \langle 10 \rangle$,[1] which is the binary representation of 2, the sum of x, y, and z.

Each of the inputs x, y, and z to the full adder has a fan-out of 3. When the operation performed by a combinational element is commutative and associative with respect to its inputs (such as the functions AND, OR, and XOR), we call the number of inputs the ***fan-in*** of the element. Although the fan-in of each gate in Figure 28.2 is 2, we could redraw the full adder to replace XOR gates A and E by a single 3-input XOR gate and OR gates F and G by a single 3-input OR gate.

To examine how the full adder operates, assume that each gate operates in unit time. Figure 28.2(a) shows a set of inputs that becomes stable at time 0. Gates A–D, and no other gates, have all their input values stable at that time and therefore produce the values shown in Figure 28.2(b) at time 1. Note that gates A–D operate in parallel. Gates E and F, but not gate G, have stable inputs at time 1 and produce the values shown in Figure 28.2(c) at time 2. The output of gate E is bit s, and so the s output from the full adder is ready at time 2. The c

[1] For clarity, we omit the commas between sequence elements when they are bits.

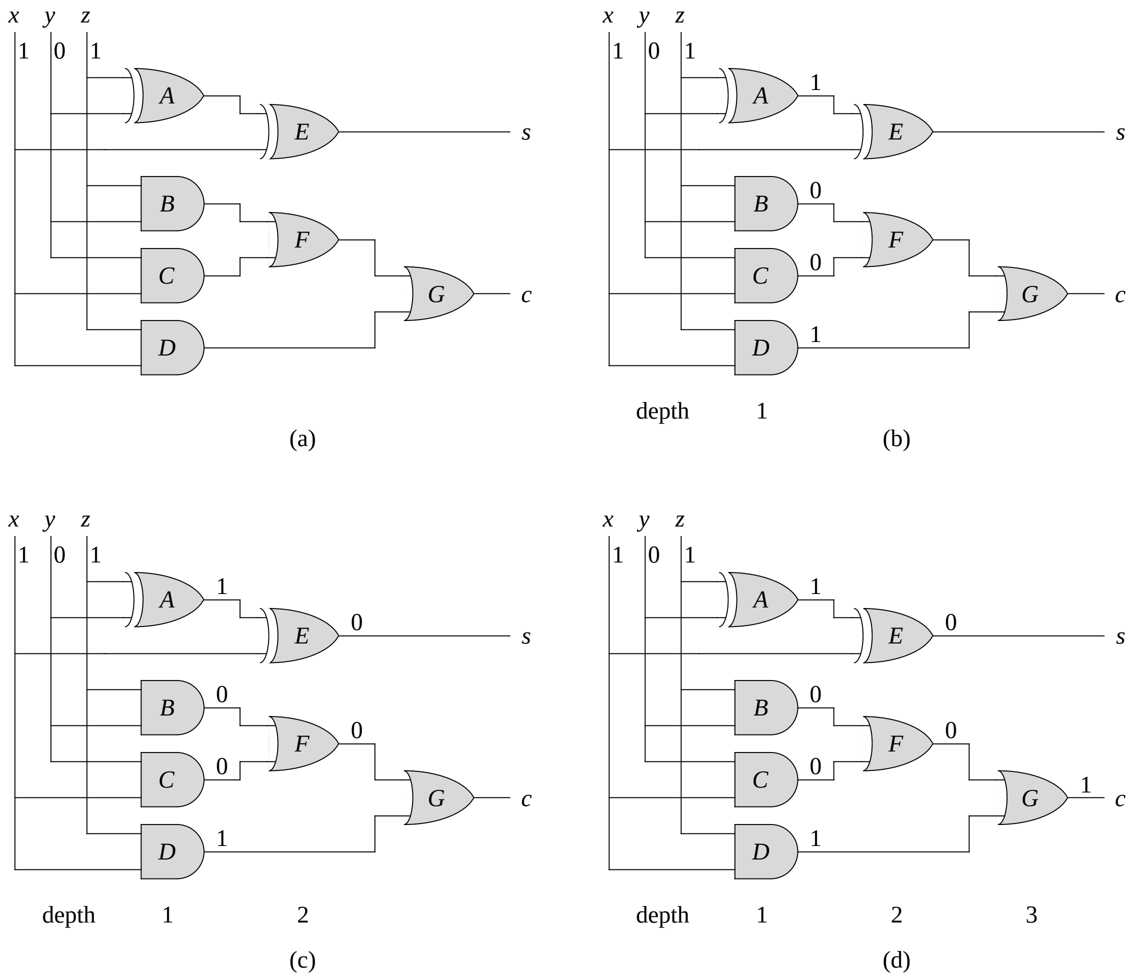


Figure 28.2 A full-adder circuit. **(a)** At time 0, the input bits shown appear on the three input wires. **(b)** At time 1, the values shown appear on the outputs of gates *A*–*D*, which are at depth 1. **(c)** At time 2, the values shown appear on the outputs of gates *E* and *F*, at depth 2. **(d)** At time 3, gate *G* produces its output, which is also the circuit output.

output is not yet ready, however. Gate *G* finally has stable inputs at time 2, and it produces the *c* output shown in Figure 28.2(d) at time 3.

Circuit depth

As in the case of the comparison networks discussed in Chapter 27, we measure the propagation delay of a combinational circuit in terms of the largest number of combinational elements on any path from the inputs to the outputs. Specifically, we define the ***depth*** of a circuit, which corresponds to its worst-case "running time," inductively in terms of the depths of its constituent wires. The depth of an input wire is 0. If a combinational element has inputs $x_1, x_2, \ldots, x_n$ at depths $d_1, d_2, \ldots, d_n$ respectively, then its outputs have depth $\max\{d_1, d_2, \ldots, d_n\} + 1$. The depth of a combinational element is the depth of its outputs. The depth of a combinational circuit is the maximum depth of any combinational element. Since we prohibit combinational circuits from containing cycles, the various notions of depth are well defined.

If each combinational element takes constant time to compute its output values, then the worst-case propagation delay through a combinational circuit is proportional to its depth. Figure 28.2 shows the depth of each gate in the full adder. Since the gate with the largest depth is gate G, the full adder itself has depth 3, which is proportional to the worst-case time it takes for the circuit to perform its function.

A combinational circuit can sometimes compute faster than its depth. Suppose that a large subcircuit feeds into one input of a 2-input AND gate but that the other input of the AND gate has value 0. The output of the gate will then be 0, independent of the input from the large subcircuit. In general, however, we cannot count on specific inputs being applied to the circuit, and the abstraction of depth as the "running time" of the circuit is therefore quite reasonable.

Circuit size

Besides circuit depth, there is another resource that we typically wish to minimize when designing circuits. The ***size*** of a combinational circuit is the number of combinational elements it contains. Intuitively, circuit size corresponds to the memory space used by an algorithm. The full adder of Figure 28.2 has size 7, for example, since it uses 7 gates.

This definition of circuit size is not particularly useful for small circuits. After all, since a full adder has a constant number of inputs and outputs and computes a well-defined function, it satisfies the definition of a combinational element. A full adder built from a single full-adder combinational element therefore has size 1. In fact, according to this definition, *any* combinational element has size 1.

The definition of circuit size is intended to apply to families of circuits that compute similar functions. For example, we shall soon see an addition circuit that takes two n-bit inputs. We are really not talking about a single circuit here, but rather a family of circuits—one for each size of input. In this context, the definition of circuit size makes good sense. It allows us to define convenient circuit elements without affecting the size of any implementation of the circuit by more than a constant factor. Of course, in practice, measurements of size are much more complicated, involving not only the choice of combinational elements, but also concerns such as the area the circuit requires when integrated on a silicon chip.

Exercises

28.1-1
In Figure 28.2, change input y to a 1. Show the resulting value carried on each wire.

28.1-2
Show how to construct an n-input parity circuit with $n - 1$ XOR gates and depth $\lceil \lg n \rceil$.

28.1-3
Show that any boolean combinational element can be constructed from a constant number of AND, OR, and NOT gates. (*Hint:* Implement the truth table for the element.)

28.1-4
Show that any boolean function can be constructed entirely out of NAND gates.

28.1-5
Construct a combinational circuit that performs the exclusive-or function using only four 2-input NAND gates.

28.1-6
Let C be an n-input, n-output combinational circuit of depth d. If two copies of C are connected, with the outputs of one feeding directly into the inputs of the other, what is the maximum possible depth of this tandem circuit? What is the minimum possible depth?

28.2 Addition circuits

We now investigate the problem of adding numbers represented in binary. We present three combinational circuits for this problem. First, we look at ripple-carry addition, which can add two n-bit numbers in $\Theta(n)$ time using a circuit with $\Theta(n)$ size. This time bound can be improved to $O(\lg n)$ using a carry-lookahead adder, which also has $\Theta(n)$ size. Finally, we present carry-save addition, which in $O(1)$ time can reduce the sum of 3 n-bit numbers to the sum of an n-bit number and an $(n + 1)$-bit number. The circuit has $\Theta(n)$ size.

8	7	6	5	4	3	2	1	0		i
1	1	0	1	1	1	0	0	0	=	c
	0	1	0	1	1	1	1	0	=	a
	1	1	0	1	0	1	0	1	=	b
1	0	0	1	1	0	0	1	1	=	s

Figure 28.3 Adding two 8-bit numbers $a = \langle 01011110 \rangle$ and $b = \langle 11010101 \rangle$ to produce a 9-bit sum $s = \langle 100110011 \rangle$. Each bit c_i is a carry bit. Each column of bits represents, from top to bottom, c_i, a_i, b_i, and s_i for some i. Carry-in c_0 is always 0.

28.2.1 Ripple-carry addition

We start with the ordinary method of summing binary numbers. We assume that a nonnegative integer a is represented in binary by a sequence of n bits $\langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$, where $n \geq \lceil \lg(a+1) \rceil$ and

$$a = \sum_{i=0}^{n-1} a_i 2^i \; .$$

Given two n-bit numbers $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ and $b = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$, we wish to produce an $(n+1)$-bit sum $s = \langle s_n, s_{n-1}, \ldots, s_0 \rangle$. Figure 28.3 shows an example of adding two 8-bit numbers. We sum columns right to left, propagating any carry from column i to column $i+1$, for $i = 0, 1, \ldots, n-1$. In the ith bit position, we take as inputs bits a_i and b_i and a ***carry-in bit*** c_i, and we produce a ***sum bit*** s_i and a ***carry-out*** bit c_{i+1}. The carry-out bit c_{i+1} from the ith position is the carry-in bit into the $(i+1)$st position. Since there is no carry-in for position 0, we assume that $c_0 = 0$. The carry-out c_n is bit s_n of the sum.

Observe that each sum bit s_i is the parity of bits a_i, b_i, and c_i (see equation (28.1)). Moreover, the carry-out bit c_{i+1} is the majority of a_i, b_i, and c_i (see equation (28.2)). Thus, each stage of the addition can be performed by a full adder.

An n-bit ***ripple-carry adder*** is formed by cascading n full adders $FA_0, FA_1, \ldots, FA_{n-1}$, feeding the carry-out c_{i+1} of FA_i directly into the carry-in input of FA_{i+1}. Figure 28.4 shows an 8-bit ripple-carry adder. The carry bits "ripple" from right to left. The carry-in c_0 to full adder FA_1 is ***hardwired*** to 0, that is, it is 0 no matter what values the other inputs take on. The output is the $(n+1)$-bit number $s = \langle s_n, s_{n-1}, \ldots, s_0 \rangle$, where s_n equals c_n, the carry-out bit from full adder FA_n.

Because the carry bits ripple through all n full adders, the time required by an n-bit ripple-carry adder is $\Theta(n)$. More precisely, full adder FA_i is at depth $i+1$ in the circuit. Because FA_{n-1} is at the largest depth of any full adder in the circuit, the depth of the ripple-carry adder is n. The size of the circuit is $\Theta(n)$ because it contains n combinational elements.

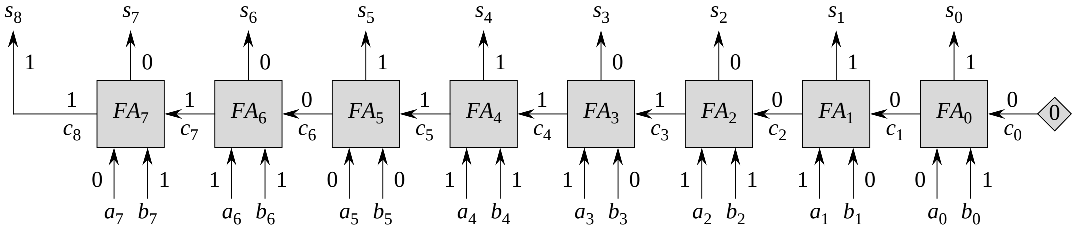

s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0
1 0 0 1 1 0 0 1 1
FA_7 FA_6 FA_5 FA_4 FA_3 FA_2 FA_1 FA_0
1 1 0 1 1 1 0 0 0 0
c_8 c_7 c_6 c_5 c_4 c_3 c_2 c_1 c_0
0 1 1 1 0 0 1 1 1 0 1 1 1 0 0 1
a_7 b_7 a_6 b_6 a_5 b_5 a_4 b_4 a_3 b_3 a_2 b_2 a_1 b_1 a_0 b_0

Figure 28.4 An 8-bit ripple-carry adder performing the addition of Figure 28.3. Carry bit c_0 is hardwired to 0, indicated by the diamond, and carry bits ripple from right to left.

a_{i-1}	b_{i-1}	c_i	carry status
0	0	0	k
0	1	c_{i-1}	p
1	0	c_{i-1}	p
1	1	1	g

Figure 28.5 The carry-out bit c_i and carry status corresponding to inputs a_{i-1}, b_{i-1}, and c_{i-1} of full adder FA_{i-1} in ripple-carry addition.

28.2.2 Carry-lookahead addition

Ripple-carry addition requires $\Theta(n)$ time because of the rippling of carry bits through the circuit. Carry-lookahead addition avoids this $\Theta(n)$-time delay by accelerating the computation of carries using a treelike circuit. A carry-lookahead adder can sum two n-bit numbers in $O(\lg n)$ time.

The key observation is that in ripple-carry addition, for $i \geq 1$, full adder FA_i has two of its input values, namely a_i and b_i, ready long before the carry-in c_i is ready. The idea behind the carry-lookahead adder is to exploit this partial information.

As an example, let $a_{i-1} = b_{i-1}$. Since the carry-out c_i is the majority function, we have $c_i = a_{i-1} = b_{i-1}$ *regardless of the carry-in* c_{i-1}. If $a_{i-1} = b_{i-1} = 0$, we can ***kill*** the carry-out c_i by forcing it to 0 without waiting for the value of c_{i-1} to be computed. Likewise, if $a_{i-1} = b_{i-1} = 1$, we can ***generate*** the carry-out $c_i = 1$, irrespective of the value of c_{i-1}.

If $a_{i-1} \neq b_{i-1}$, however, then c_i depends on c_{i-1}. Specifically, $c_i = c_{i-1}$, because the carry-in c_{i-1} casts the deciding "vote" in the majority election that determines c_i. In this case, we ***propagate*** the carry, since the carry-out is the carry-in.

Figure 28.5 summarizes these relationships in terms of ***carry statuses***, where k is "carry kill," g is "carry generate," and p is "carry propagate."

		FA_i		
	⊗	k	p	g
FA_{i-1}	k	k	k	g
	p	k	p	g
	g	k	g	g

Figure 28.6 The carry status of the combination of full adders FA_{i-1} and FA_i in terms of their individual carry statuses, given by the carry-status operator $\otimes$ over the domain $\{\mathsf{k}, \mathsf{p}, \mathsf{g}\}$.

Consider two consecutive full adders FA_{i-1} and FA_i together as a combined unit. The carry-in to the unit is c_{i-1}, and the carry-out is c_{i+1}. We can view the combined unit as killing, generating, or propagating carries, much as for a single full adder. The combined unit kills its carry if FA_i kills its carry or if FA_{i-1} kills its carry and FA_i propagates it. Similarly, the combined unit generates a carry if FA_i generates a carry or if FA_{i-1} generates a carry and FA_i propagates it. The combined unit propagates the carry, setting $c_{i+1} = c_{i-1}$, if both full adders propagate carries. The table in Figure 28.6 summarizes how carry statuses are combined when full adders are juxtaposed. We can view this table as the definition of the ***carry-status operator*** $\otimes$ over the domain $\{\mathsf{k}, \mathsf{p}, \mathsf{g}\}$. An important property of this operator is that it is associative, as Exercise 28.2-2 asks you to verify.

We can use the carry-status operator to express each carry bit c_i in terms of the inputs. We start by defining $x_0 = \mathsf{k}$ and

$$x_i = \begin{cases} \mathsf{k} & \text{if } a_{i-1} = b_{i-1} = 0 \,, \\ \mathsf{p} & \text{if } a_{i-1} \neq b_{i-1} \,, \\ \mathsf{g} & \text{if } a_{i-1} = b_{i-1} = 1 \,, \end{cases} \tag{28.3}$$

for $i = 1, 2, \ldots, n$. Thus, for $i = 1, 2, \ldots, n$, the value of x_i is the carry status given by Figure 28.5.

The carry-out c_i of a given full adder FA_{i-1} can depend on the carry status of every full adder FA_j for $j = 0, 1, \ldots, i-1$. Let us define $y_0 = x_0 = \mathsf{k}$ and

$$\begin{aligned} y_i &= y_{i-1} \otimes x_i & (28.4) \\ &= x_0 \otimes x_1 \otimes \cdots \otimes x_i \end{aligned}$$

for $i = 1, 2, \ldots, n$. We can think of y_i as a "prefix" of the "product" $x_0 \otimes x_1 \otimes \cdots \otimes x_n$; we call the process of computing the values $y_0, y_1, \ldots, y_n$ a ***prefix computation***. (Chapter **??** discusses prefix computations in a more general parallel context.) Figure 28.7 shows the values of x_i and y_i corresponding to the binary addition shown in Figure 28.3. The following lemma gives the significance of the y_i values for carry-lookahead addition.

i	8	7	6	5	4	3	2	1	0
a_i		0	1	0	1	1	1	1	0
b_i		1	1	0	1	0	1	0	1
x_i	p	g	k	g	p	g	p	p	k
y_i	g	g	k	g	g	g	k	k	k
c_i	1	1	0	1	1	1	0	0	0

Figure 28.7 The values of x_i and y_i for $i = 0, 1, \ldots, 8$ that correspond to the values of a_i, b_i, and c_i in the binary-addition problem of Figure 28.3. Each value of x_i is shaded with the values of a_{i-1} and b_{i-1} that it depends on.

Lemma 28.1

Define $x_0, x_1, \ldots, x_n$ and $y_0, y_1, \ldots, y_n$ by equations (28.3) and (28.4). For $i = 0, 1, \ldots, n$, the following conditions hold:

1. $y_i = \mathsf{k}$ implies $c_i = 0$,
2. $y_i = \mathsf{g}$ implies $c_i = 1$, and
3. $y_i = \mathsf{p}$ does not occur.

Proof The proof is by induction on i. For the basis, $i = 0$. We have $y_0 = x_0 = \mathsf{k}$ by definition, and also $c_0 = 0$. For the inductive step, assume that the lemma holds for $i - 1$. There are three cases depending on the value of y_i.

1. If $y_i = \mathsf{k}$, then since $y_i = y_{i-1} \otimes x_i$, the definition of the carry-status operator $\otimes$ from Figure 28.6 implies either that $x_i = \mathsf{k}$ or that $x_i = \mathsf{p}$ and $y_{i-1} = \mathsf{k}$. If $x_i = \mathsf{k}$, then equation (28.3) implies that $a_{i-1} = b_{i-1} = 0$, and thus $c_i = \text{majority}(a_{i-1}, b_{i-1}, c_{i-1}) = 0$. If $x_i = \mathsf{p}$ and $y_{i-1} = \mathsf{k}$, then $a_{i-1} \neq b_{i-1}$ and, by induction, $c_{i-1} = 0$. Thus, $\text{majority}(a_{i-1}, b_{i-1}, c_{i-1}) = 0$, and thus $c_i = 0$.
2. If $y_i = \mathsf{g}$, then either we have $x_i = \mathsf{g}$ or we have $x_i = \mathsf{p}$ and $y_{i-1} = \mathsf{g}$. If $x_i = \mathsf{g}$, then $a_{i-1} = b_{i-1} = 1$, which implies $c_i = 1$. If $x_i = \mathsf{p}$ and $y_{i-1} = \mathsf{g}$, then $a_{i-1} \neq b_{i-1}$ and, by induction, $c_{i-1} = 1$, which implies $c_i = 1$.
3. If $y_i = \mathsf{p}$, then Figure 28.6 implies that $y_{i-1} = \mathsf{p}$, which contradicts the inductive hypothesis. ■

Lemma 28.1 implies that we can compute each carry bit c_i by computing each carry status y_i. Once we have all the carry bits, we can compute the entire sum in $\Theta(1)$ time by computing in parallel the sum bits $s_i = \text{parity}(a_i, b_i, c_i)$ for $i = 0, 1, \ldots, n$ (taking $a_n = b_n = 0$). Thus, the problem of quickly adding two numbers reduces to the prefix computation of the carry statuses $y_0, y_1, \ldots, y_n$.

Computing carry statuses with a parallel prefix circuit

By using a prefix circuit that operates in parallel, as opposed to a ripple-carry circuit that produces its outputs one by one, we can compute all n carry statuses $y_0, y_1, \ldots, y_n$ more quickly. Specifically, we shall design a parallel prefix circuit with $O(\lg n)$ depth. The circuit has $\Theta(n)$ size—asymptotically the same amount of hardware as a ripple-carry adder.

Before constructing the parallel prefix circuit, we introduce a notation that will aid our understanding of how the circuit operates. For integers i and j in the range $0 \le i \le j \le n$, we define

$$[i, j] = x_i \otimes x_{i+1} \otimes \cdots \otimes x_j \ .$$

Thus, for $i = 0, 1, \ldots, n$, we have $[i, i] = x_i$, since the composition of just one carry status x_i is itself. For i, j, and k satisfying $0 \le i < j \le k \le n$, we also have the identity

$$[i, k] = [i, j - 1] \otimes [j, k] \ , \tag{28.5}$$

since the carry-status operator is associative. The goal of a prefix computation, in terms of this notation, is to compute $y_i = [0, i]$ for $i = 0, 1, \ldots, n$.

The only combinational element used in the parallel prefix circuit is a circuit that computes the $\otimes$ operator. Figure 28.8 shows how pairs of $\otimes$ elements are organized to form the internal nodes of a complete binary tree, and Figure 28.9 illustrates the parallel prefix circuit for $n = 8$. Note that the wires in the circuit follow the structure of a tree, but the circuit itself is not a tree, although it is purely combinational. The inputs $x_1, x_2, \ldots, x_n$ are supplied at the leaves, and the input x_0 is provided at the root. The outputs $y_0, y_1, \ldots, y_{n-1}$ are produced at leaves, and the output y_n is produced at the root. (For ease in understanding the prefix computation, variable indices increase from left to right in Figures 28.8 and 28.9, rather than from right to left as in other figures of this section.)

The two $\otimes$ elements in each node typically operate at different times and have different depths in the circuit. As shown in Figure 28.8, if the subtree rooted at a given node spans some range $x_i, x_{i+1}, \ldots, x_k$ of inputs, its left subtree spans the range $x_i, x_{i+1}, \ldots, x_{j-1}$, and its right subtree spans the range $x_j, x_{j+1}, \ldots, x_k$, then the node must produce for its parent the product $[i, k]$ of all inputs spanned by its subtree. Since we can assume inductively that the node's left and right children produce the products $[i, j - 1]$ and $[j, k]$, the node simply uses one of its two elements to compute $[i, k] \leftarrow [i, j - 1] \otimes [j, k]$.

Some time after this upward phase of computation, the node receives from its parent the product $[0, i - 1]$ of all inputs that come before the leftmost input x_i that it spans. The node now likewise computes values for its children. The leftmost input spanned by the node's left child is also x_i, and so it passes the value $[0, i - 1]$ to the left child unchanged. The leftmost input spanned by its right child is x_j,

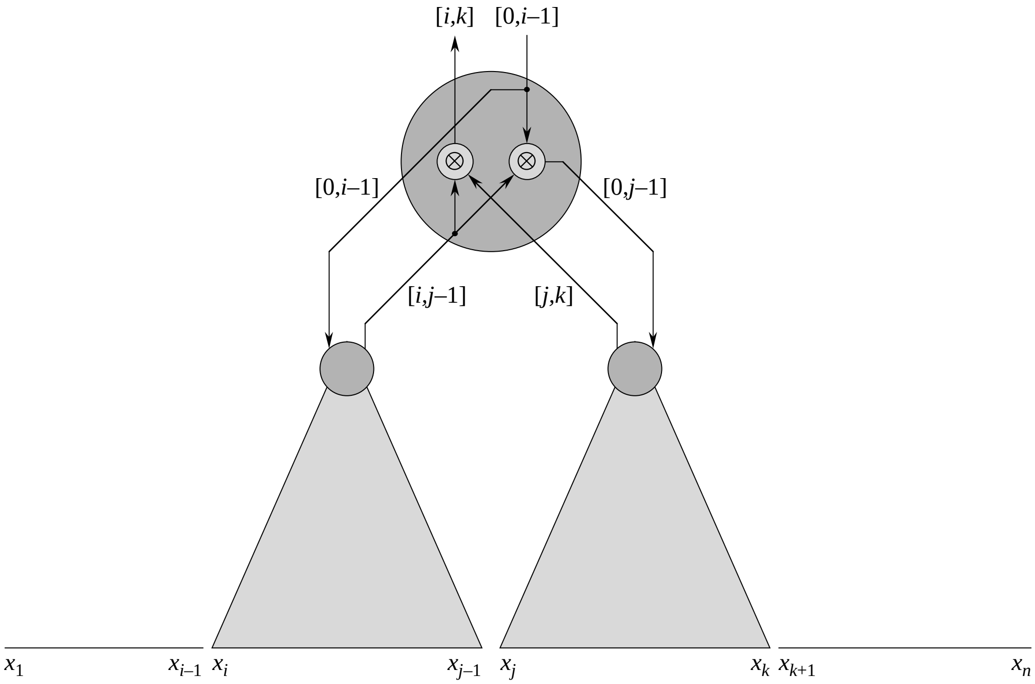


Figure 28.8 The organization of a parallel prefix circuit. The node shown is the root of a subtree whose leaves input the values x_i to x_k. The node's left subtree spans inputs x_i to x_{j-1}, and its right subtree spans inputs x_j to x_k. The node consists of two $\otimes$ elements, which operate at different times during the operation of the circuit. One element computes $[i, k] \leftarrow [i, j-1] \otimes [j, k]$, and the other element computes $[0, j-1] \leftarrow [0, i-1] \otimes [i, j-1]$. The values computed are shown on the wires.

and so it must produce $[0, j-1]$. Since the node receives the value $[0, i-1]$ from its parent and the value $[i, j-1]$ from its left child, it simply computes $[0, j-1] \leftarrow [0, i-1] \otimes [i, j-1]$ and sends this value to the right child.

Figure 28.9 shows the resulting circuit, including the boundary case that arises at the root. The value $x_0 = [0, 0]$ is provided as input at the root, and one more $\otimes$ element is used to compute (in general) the value $y_n = [0, n] = [0, 0] \otimes [1, n]$.

If n is an exact power of 2, then the parallel prefix circuit uses $2n - 1$ $\otimes$ elements. It takes only $O(\lg n)$ time to compute all $n + 1$ prefixes, since the computation proceeds up the tree and then back down. Exercise 28.2-5 studies the depth of the circuit in more detail.

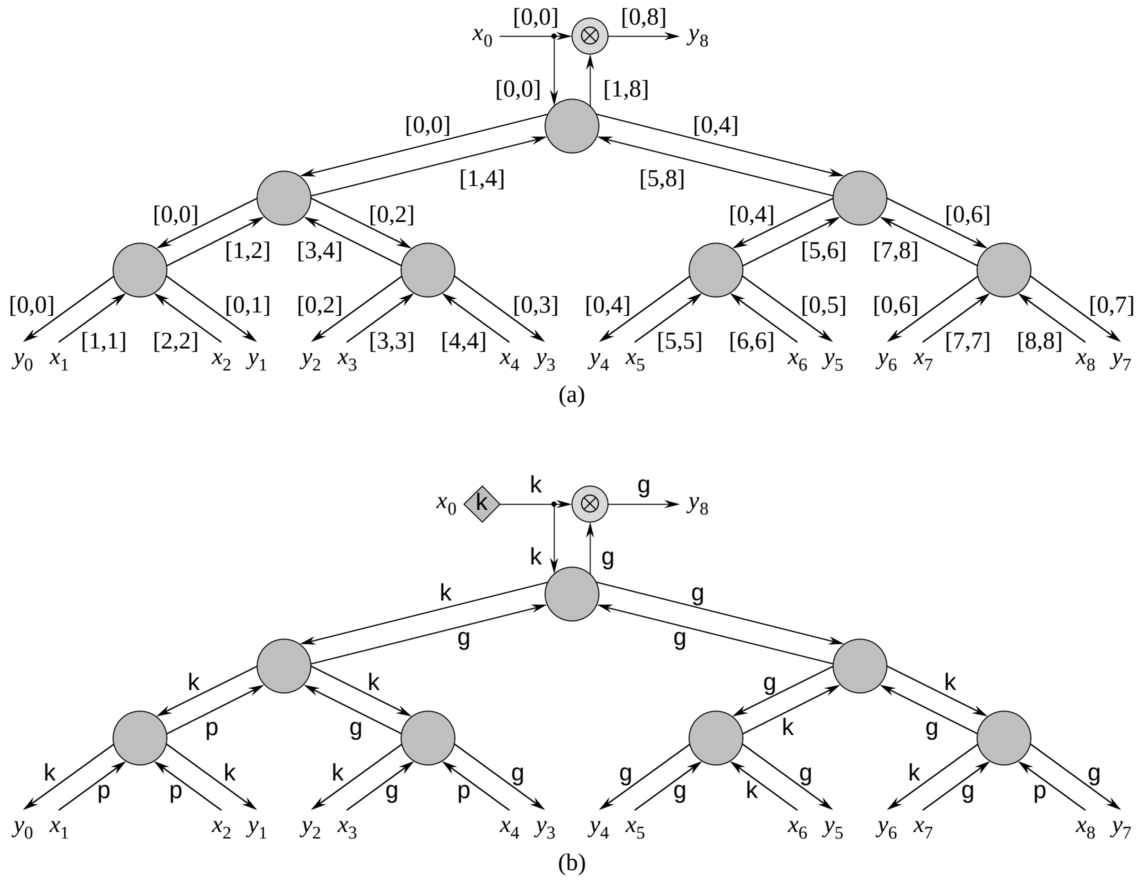


Figure 28.9 A parallel prefix circuit for $n = 8$. **(a)** The overall structure of the circuit, and the values carried on each wire. **(b)** The same circuit with values corresponding to Figures 28.3 and 28.7.

Completing the carry-lookahead adder

Now that we have a parallel prefix circuit, we can complete the description of the carry-lookahead adder. Figure 28.10 shows the construction. An n-bit ***carry-lookahead adder*** consists of $n + 1$ ***KPG boxes***, each of $\Theta(1)$ size, and a parallel prefix circuit with inputs $x_0, x_1, \ldots, x_n$ (x_0 is hardwired to k) and outputs $y_0, y_1, \ldots, y_n$. KPG box KPG_i takes external inputs a_i and b_i and produces sum bit s_i. (Input bits a_n and b_n are hardwired to 0.) Given a_{i-1} and b_{i-1}, box KPG_{i-1} computes $x_i \in \{\mathsf{k}, \mathsf{p}, \mathsf{g}\}$ according to equation (28.3) and sends this value as the external input x_i of the parallel prefix circuit. (The value of x_{n+1} is ignored.) Computing all the x_i takes $\Theta(1)$ time. After a delay of $O(\lg n)$, the parallel prefix circuit

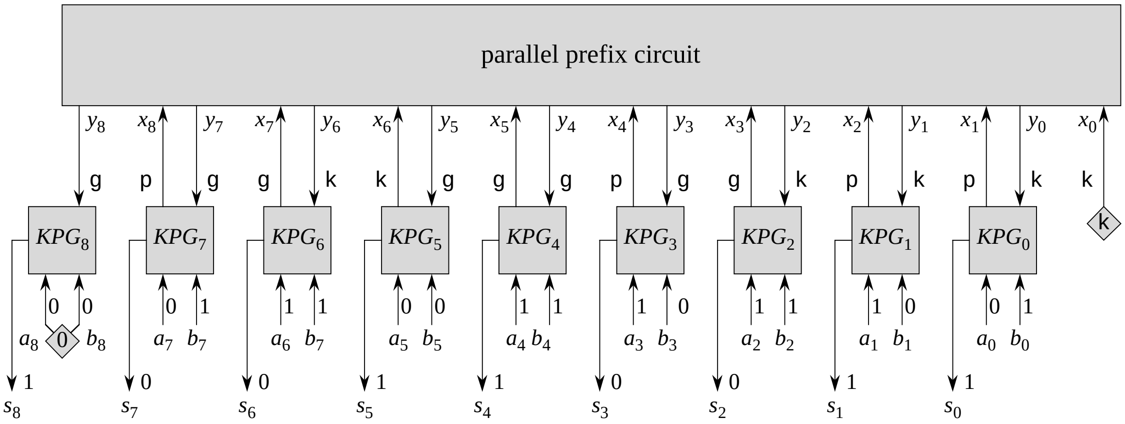

parallel prefix circuit
y8 x8 y7 x7 y6 x6 y5 x5 y4 x4 y3 x3 y2 x2 y1 x1 y0 x0
g p g g k k g g g p g g k p k p k k
KPG8 KPG7 KPG6 KPG5 KPG4 KPG3 KPG2 KPG1 KPG0
k
0 0 0 1 1 1 0 0 1 1 1 0 1 1 1 0 0 1
a8 0 b8 a7 b7 a6 b7 a5 b5 a4 b4 a3 b3 a2 b2 a1 b1 a0 b0
1 0 0 1 1 0 0 1 1
s8 s7 s6 s5 s4 s3 s2 s1 s0

Figure 28.10 The construction of an n-bit carry-lookahead adder, shown here for $n = 8$. It consists of $n+1$ KPG boxes KPG_i for $i = 0, 1, \ldots, n$. Each box KPG_i takes external inputs a_i and b_i (where a_n and b_n are hardwired to 0, as indicated by the diamond) and computes carry status x_{i+1}. These values are fed into the parallel prefix circuit, which returns the results y_i of the prefix computation. Each box KPG_i now takes y_i as input, interprets it as the carry-in bit c_i, and then outputs the sum bit $s_i = \text{parity}(a_i, b_i, c_i)$. Sample values corresponding to those shown in Figures 28.3 and 28.9 are shown.

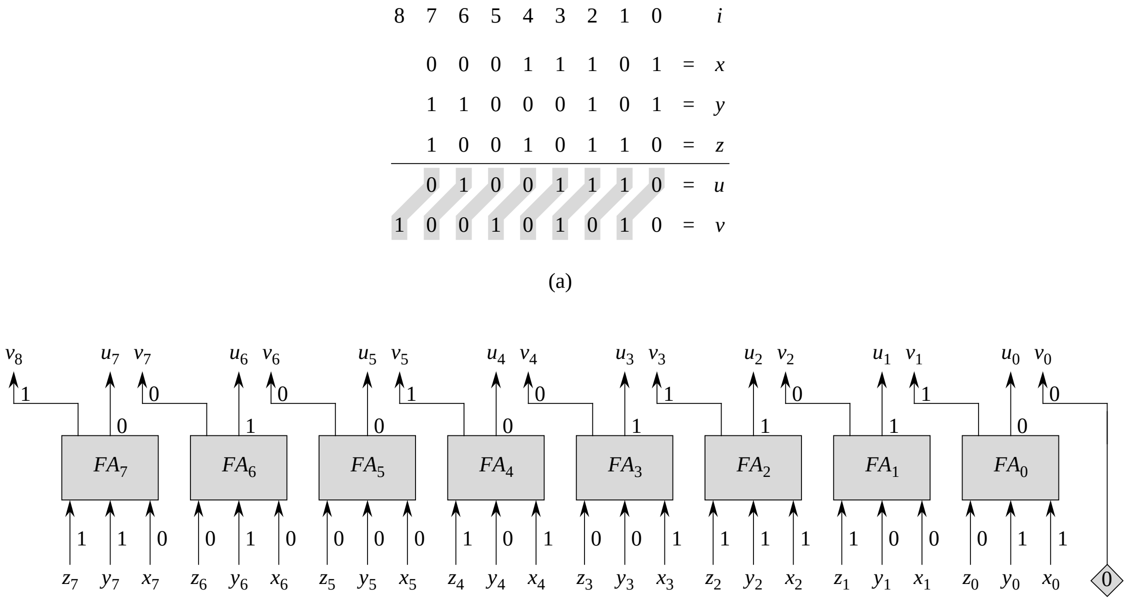
produces $y_0, y_1, \ldots, y_n$. By Lemma 28.1, y_i is either k or g; it cannot be p. Each value y_i indicates the carry-in to full adder FA_i in the ripple-carry adder: $y_i = \mathsf{k}$ implies $c_i = 0$, and $y_i = \mathsf{g}$ implies $c_i = 1$. Thus, the value of y_i is fed into KPG_i to indicate the carry-in c_i, and the sum bit $s_i = \text{parity}(a_i, b_i, c_i)$ is produced in constant time. Thus, the carry-lookahead adder operates in $O(\lg n)$ time and has $\Theta(n)$ size.

28.2.3 Carry-save addition

A carry-lookahead adder can add two n-bit numbers in $O(\lg n)$ time. Perhaps surprisingly, adding three n-bit numbers takes only a constant additional amount of time. The trick is to reduce the problem of adding three numbers to the problem of adding just two numbers.

Given three n-bit numbers $x = \langle x_{n-1}, x_{n-2}, \ldots, x_0 \rangle$, $y = \langle y_{n-1}, y_{n-2}, \ldots, y_0 \rangle$, and $z = \langle z_{n-1}, z_{n-2}, \ldots, z_0 \rangle$, an n-bit ***carry-save adder*** produces an n-bit number $u = \langle u_{n-1}, u_{n-2}, \ldots, u_0 \rangle$ and an $(n + 1)$-bit number $v = \langle v_n, v_{n-1}, \ldots, v_0 \rangle$ such that

$$u + v = x + y + z \; .$$


8 7 6 5 4 3 2 1 0 i
0 0 0 1 1 1 0 1 = x
1 1 0 0 0 1 0 1 = y
1 0 0 1 0 1 1 0 = z
0 1 0 0 1 1 1 0 = u
1 0 0 1 0 1 0 1 0 = v
v8 u7 v7 u6 v6 u5 v5 u4 v4 u3 v3 u2 v2 u1 v1 u0 v0
FA7 FA6 FA5 FA4 FA3 FA2 FA1 FA0
z7 y7 x7 z6 y6 x6 z5 y5 x5 z4 y4 x4 z3 y3 x3 z2 y2 x2 z1 y1 x1 z0 y0 x0
0

(a)

(b)

Figure 28.11 **(a)** Carry-save addition. Given three n-bit numbers x, y, and z, we produce an n-bit number u and an $(n+1)$-bit number v such that $x + y + z = u + v$. The ith pair of shaded bits are a function of x_i, y_i, and z_i. **(b)** An 8-bit carry-save adder. Each full adder FA_i takes inputs x_i, y_i, and z_i and produces sum bit u_i and carry-out bit v_{i+1}. Bit v_0 is hardwired to 0.

As shown in Figure 28.11(a), it does this by computing

$$\begin{aligned} u_i &= \text{parity}(x_i, y_i, z_i) , \\ v_{i+1} &= \text{majority}(x_i, y_i, z_i) , \end{aligned}$$

for $i = 0, 1, \ldots, n-1$. Bit v_0 always equals 0.

The n-bit carry-save adder shown in Figure 28.11(b) consists of n full adders $FA_0, FA_1, \ldots, FA_{n-1}$. For $i = 0, 1, \ldots, n-1$, full adder FA_i takes inputs x_i, y_i, and z_i. The sum-bit output of FA_i is taken as u_i, and the carry-out of FA_i is taken as v_{i+1}. Bit v_0 is hardwired to 0.

Since the computations of all $2n+1$ output bits are independent, they can be performed in parallel. Thus, a carry-save adder operates in $\Theta(1)$ time and has $\Theta(n)$ size. To sum three n-bit numbers, therefore, we need only perform a carry-save addition, taking $\Theta(1)$ time, and then perform a carry-lookahead addition, taking $O(\lg n)$ time. Although this method is not asymptotically better than the method of using two carry-lookahead additions, it is much faster in practice. Moreover, we shall see in Section 28.3 that carry-save addition is central to fast algorithms for multiplication.

Exercises

28.2-1

Let $a = \langle 01111111 \rangle$, $b = \langle 00000001 \rangle$, and $n = 8$. Show the sum and carry bits output by full adders when ripple-carry addition is performed on these two sequences. Show the carry statuses $x_0, x_1, \ldots, x_8$ corresponding to a and b, label each wire of the parallel prefix circuit of Figure 28.9 with the value it has given these x_i inputs, and show the resulting outputs $y_0, y_1, \ldots, y_8$.

28.2-2

Prove that the carry-status operator $\otimes$ given by Figure 28.5 is associative.

28.2-3

Show by example how to construct an $O(\lg n)$-time parallel prefix circuit for values of n that are not exact powers of 2 by drawing a parallel prefix circuit for $n = 11$. Characterize the performance of parallel prefix circuits built in the shape of arbitrary binary trees.

28.2-4

Show the gate-level construction of the box KPG_i. Assume that each output x_i is represented by $\langle 00 \rangle$ if $x_i = \mathsf{k}$, by $\langle 11 \rangle$ if $x_i = \mathsf{g}$, and by $\langle 01 \rangle$ or $\langle 10 \rangle$ if $x_i = \mathsf{p}$. Assume also that each input y_i is represented by 0 if $y_i = \mathsf{k}$ and by 1 if $y_i = \mathsf{g}$.

28.2-5

Label each wire in the parallel prefix circuit of Figure 28.9(a) with its depth. A ***critical path*** in a circuit is a path with the largest number of combinational elements on any path from inputs to outputs. Identify the critical path in Figure 28.9(a), and show that its length is $O(\lg n)$. Show that some node has $\otimes$ elements that operate $\Theta(\lg n)$ time apart. Is there a node whose $\otimes$ elements operate simultaneously?

28.2-6

Give a recursive block diagram of the circuit in Figure 28.12 for any number n of inputs that is an exact power of 2. Argue on the basis of your block diagram that the circuit indeed performs a prefix computation. Show that the depth of the circuit is $\Theta(\lg n)$ and that it has $\Theta(n \lg n)$ size.

28.2-7

A ***tally circuit*** has n binary inputs and $m = \lceil \lg(n+1) \rceil$ outputs. Interpreted as a binary number, the outputs give the number of 1's in the inputs. For example, if

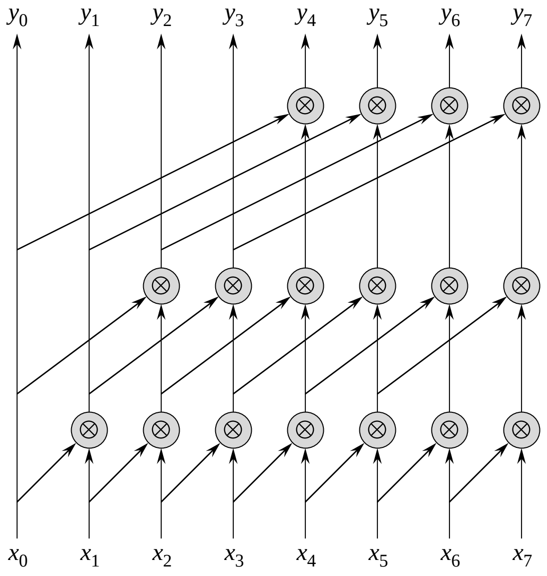


Figure 28.12 A parallel prefix circuit for use in Exercise 28.2-6.

the input is $\langle 10011110 \rangle$, the output is $\langle 101 \rangle$, indicating that there are five 1's in the input. Describe an $O(\lg n)$-depth tally circuit having $\Theta(n)$ size.

28.2-8 ★

Show that n-bit addition can be accomplished with a combinational circuit of depth 4 and size polynomial in n if AND and OR gates are allowed arbitrarily high fan-in. (*Optional:* Achieve depth 3.)

28.2-9 ★

Suppose that two random n-bit numbers are added with a ripple-carry adder, where each bit is independently 0 or 1 with equal probability. Show that with probability at least $1 - 1/n$, no carry propagates farther than $O(\lg n)$ consecutive stages. In other words, although the depth of the ripple-carry adder is $\Theta(n)$, for two random numbers, the outputs almost always settle within $O(\lg n)$ time.

28.3 Multiplication circuits

The "grade-school" multiplication algorithm in Figure 28.13 can compute the $2n$-bit product $p = \langle p_{2n-1}, p_{2n-2}, \ldots, p_0 \rangle$ of two n-bit numbers $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ and $b = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$. We examine the bits of b, from b_0 up to

$$\begin{array}{cccccccccc}
 & & & & 1 & 1 & 1 & 0 & = & a \\
 & & & & 1 & 1 & 0 & 1 & = & b \\
\hline
 & & & & 1 & 1 & 1 & 0 & = & m^{(0)} \\
 & & & 0 & 0 & 0 & 0 & & = & m^{(1)} \\
 & & 1 & 1 & 1 & 0 & & & = & m^{(2)} \\
 & 1 & 1 & 1 & 0 & & & & = & m^{(3)} \\
\hline
1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & = & p
\end{array}$$

Figure 28.13 The "grade-school" multiplication method, shown here multiplying $a = \langle 1110 \rangle$ by $b = \langle 1101 \rangle$ to obtain the product $p = \langle 10110110 \rangle$. We add $\sum_{i=0}^{n-1} m^{(i)}$, where $m^{(i)} = a \cdot b_i \cdot 2^i$. Here, $n = 4$. Each term $m^{(i)}$ is formed by shifting either a (if $b_i = 1$) or 0 (if $b_i = 0$) i positions to the left. Bits that are not shown are 0 regardless of the values of a and b.

b_{n-1}. For each bit b_i with a value of 1, we add a into the product, but shifted left by i positions. For each bit b_i with a value of 0, we add in 0. Thus, letting $m^{(i)} = a \cdot b_i \cdot 2^i$, we compute

$$p = a \cdot b = \sum_{i=0}^{n-1} m^{(i)} .$$

Each term $m^{(i)}$ is called a ***partial product***. There are n partial products to sum, with bits in positions 0 to $2n - 2$. The carry-out from the highest bit yields the final bit in position $2n - 1$.

In this section, we examine two circuits for multiplying two n-bit numbers. Array multipliers operate in $\Theta(n)$ time and have $\Theta(n^2)$ size. Wallace-tree multipliers also have $\Theta(n^2)$ size, but they operate in $\Theta(\lg n)$ time. Both circuits are based on the grade-school algorithm.

28.3.1 Array multipliers

An array multiplier consists conceptually of three parts. The first part forms the partial products. The second sums the partial products using carry-save adders. Finally, the third sums the two numbers resulting from the carry-save additions using either a ripple-carry or carry-lookahead adder.

Figure 28.14 shows an ***array multiplier*** for two input numbers $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ and $b = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$. The a_j values run vertically, and the b_i values run horizontally. Each input bit fans out to n AND gates to form partial products. Full adders, which are organized as carry-save adders, sum partial products. The lower-order bits of the final product are output on the right. The higher-order bits are formed by adding the two numbers output by the last carry-save adder.

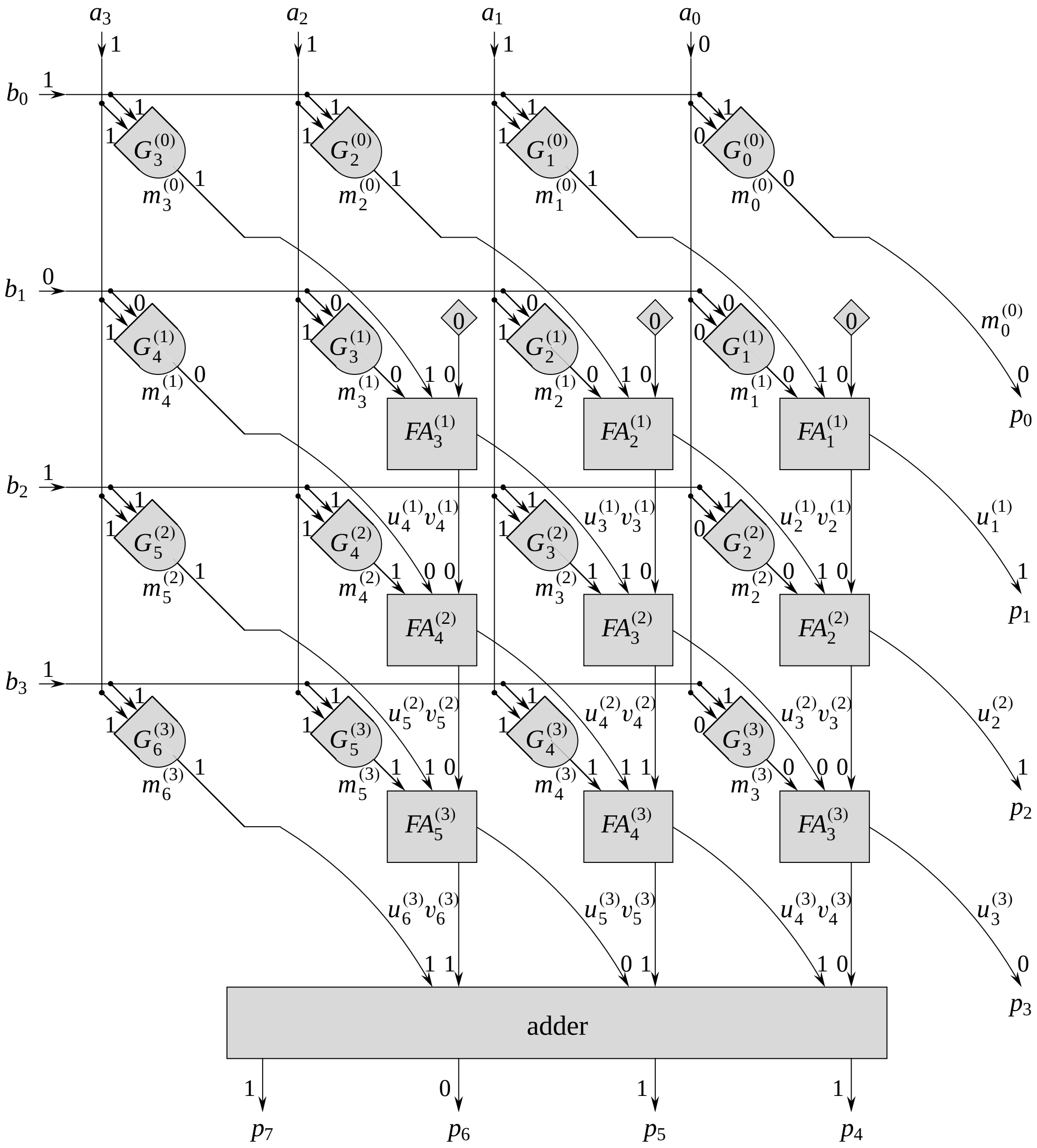


Figure 28.14 An array multiplier that computes the product $p = \langle p_{2n-1}, p_{2n-2}, \ldots, p_0 \rangle$ of two n-bit numbers $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ and $b = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$, shown here for $n = 4$. Each AND gate $G_j^{(i)}$ computes partial-product bit $m_j^{(i)}$. Each row of full adders constitutes a carry-save adder. The lower n bits of the product are $m_0^{(0)}$ and the u bits coming out from the rightmost column of full adders. The upper n product bits are formed by adding the u and v bits coming out from the bottom row of full adders. Shown are bit values for inputs $a = \langle 1110 \rangle$ and $b = \langle 1101 \rangle$ and product $p = \langle 10110110 \rangle$, corresponding to Figures 28.13 and 28.15.

Let us examine the construction of the array multiplier more closely. Given the two input numbers $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ and $b = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$, the bits of the partial products are easy to compute. Specifically, for $i, j = 0, 1, \ldots, n-1$, we have

$$m^{(i)}_{j+i} = a_j \cdot b_i \ .$$

Since the product of 1-bit values can be computed directly with an AND gate, all the bits of the partial products (except those known to be 0, which need not be explicitly computed) can be produced in one step using n^2 AND gates.

Figure 28.15 illustrates how the array multiplier performs the carry-save additions when summing the partial products in Figure 28.13. It starts by carry-save adding $m^{(0)}$, $m^{(1)}$, and 0, yielding an $(n+1)$-bit number $u^{(1)}$ and an $(n+1)$-bit number $v^{(1)}$. (The number $v^{(1)}$ has only $n+1$ bits, not $n+2$, because the $(n+1)$st bits of both 0 and $m^{(0)}$ are 0.) Thus, $m^{(0)} + m^{(1)} = u^{(1)} + v^{(1)}$. It then carry-save adds $u^{(1)}$, $v^{(1)}$, and $m^{(2)}$, yielding an $(n+2)$-bit number $u^{(2)}$ and an $(n+2)$-bit number $v^{(2)}$. (Again, $v^{(2)}$ has only $n+2$ bits because both $u^{(1)}_{n+2}$ and $v^{(1)}_{n+2}$ are 0.) We then have $m^{(0)} + m^{(1)} + m^{(2)} = u^{(2)} + v^{(2)}$. The multiplier continues on, carry-save adding $u^{(i-1)}$, $v^{(i-1)}$, and $m^{(i)}$ for $i = 2, 3, \ldots, n-1$. The result is a $(2n-1)$-bit number $u^{(n-1)}$ and a $(2n-1)$-bit number $v^{(n-1)}$, where

$$\begin{aligned} u^{(n-1)} + v^{(n-1)} &= \sum_{i=0}^{n-1} m^{(i)} \\ &= p \ . \end{aligned}$$

In fact, the carry-save additions in Figure 28.15 operate on more bits than strictly necessary. Observe that for $i = 1, 2, \ldots, n-1$ and $j = 0, 1, \ldots, i-1$, we have $m^{(i)}_j = 0$ because of how we shift the partial products. Observe also that $v^{(i)}_j = 0$ for $i = 1, 2, \ldots, n-1$ and $j = 0, 1, \ldots, i, i+n, i+n+1, \ldots, 2n-1$. (See Exercise 28.3-1.) Each carry-save addition, therefore, needs to operate on only $n-1$ bits.

Let us now examine the correspondence between the array multiplier and the repeated carry-save addition scheme. Each AND gate is labeled by $G^{(i)}_j$ for some i and j in the ranges $0 \le i \le n-1$ and $0 \le j \le 2n-2$. Gate $G^{(i)}_j$ produces $m^{(i)}_j$, the jth bit of the ith partial product. For $i = 0, 1, \ldots, n-1$, the ith row of AND gates computes the n significant bits of the partial product $m^{(i)}$, that is, $\langle m^{(i)}_{n+i-1}, m^{(i)}_{n+i-2}, \ldots, m^{(i)}_i \rangle$.

Except for the full adders in the top row (that is, for $i = 2, 3, \ldots, n-1$), each full adder $FA^{(i)}_j$ takes three input bits—$m^{(i)}_j$, $u^{(i-1)}_j$, and $v^{(i-1)}_j$—and produces two output bits—$u^{(i)}_j$ and $v^{(i)}_{j+1}$. (Note that in the leftmost column of full adders, $u^{(i-1)}_{i+n-1} = m^{(i)}_{i+n-1}$.) Each full adder $FA^{(1)}_j$ in the top row takes inputs $m^{(0)}_j$, $m^{(1)}_j$, and 0 and produces bits $u^{(1)}_j$ and $v^{(1)}_{j+1}$.

$$
\begin{array}{cccccccccl}
 & & & & 0 & 0 & 0 & 0 & = & 0 \\
 & & & & 1 & 1 & 1 & 0 & = & m^{(0)} \\
 & & & 0 & 0 & 0 & 0 & & = & m^{(1)} \\
\hline
 & & & 0 & 1 & 1 & 1 & 0 & = & u^{(1)} \\
 & & & 0 & 0 & 0 & & & = & v^{(1)} \\
 & & 1 & 1 & 1 & 0 & & & = & m^{(2)} \\
\hline
 & & 1 & 1 & 0 & 1 & 1 & 0 & = & u^{(2)} \\
 & & 0 & 1 & 0 & & & & = & v^{(2)} \\
 & 1 & 1 & 1 & 0 & & & & = & m^{(3)} \\
\hline
 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & = & u^{(3)} \\
 & 1 & 1 & 0 & & & & & = & v^{(3)} \\
\hline
1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & = & p
\end{array}
$$

Figure 28.15 Evaluating the sum of the partial products by repeated carry-save addition. For this example, $a = \langle 1110 \rangle$ and $b = \langle 1101 \rangle$. Bits that are blank are 0 regardless of the values of a and b. We first evaluate $m^{(0)} + m^{(1)} + 0 = u^{(1)} + v^{(1)}$, then $u^{(1)} + v^{(1)} + m^{(2)} = u^{(2)} + v^{(2)}$, then $u^{(2)} + v^{(2)} + m^{(3)} = u^{(3)} + v^{(3)}$, and finally $p = m^{(0)} + m^{(1)} + m^{(2)} + m^{(3)} = u^{(3)} + v^{(3)}$. Note that $p_0 = m_0^{(0)}$ and $p_i = u_i^{(i)}$ for $i = 1, 2, \ldots, n-1$.

Finally, let us examine the output of the array multiplier. As we observed above, $v_j^{(n-1)} = 0$ for $j = 0, 1, \ldots, n-1$. Thus, $p_j = u_j^{(n-1)}$ for $j = 0, 1, \ldots, n-1$. Moreover, since $m_0^{(1)} = 0$, we have $u_0^{(1)} = m_0^{(0)}$, and since the lowest-order i bits of each $m^{(i)}$ and $v^{(i-1)}$ are 0, we have $u_j^{(i)} = u_j^{(i-1)}$ for $i = 2, 3, \ldots, n-1$ and $j = 0, 1, \ldots, i-1$. Thus, $p_0 = m_0^{(0)}$ and, by induction, $p_i = u_i^{(i)}$ for $i = 1, 2, \ldots, n-1$. Product bits $\langle p_{2n-1}, p_{2n-2}, \ldots, p_n \rangle$ are produced by an n-bit adder that adds the outputs from the last row of full adders:

$$
\begin{aligned}
&\langle p_{2n-1}, p_{2n-2}, \ldots, p_n \rangle = \\
&\qquad \langle u_{2n-2}^{(n-1)}, u_{2n-3}^{(n-1)}, \ldots, u_n^{(n-1)} \rangle + \langle v_{2n-2}^{(n-1)}, v_{2n-3}^{(n-1)}, \ldots, v_n^{(n-1)} \rangle .
\end{aligned}
$$

Analysis

Data ripple through an array multiplier from upper left to lower right. It takes $\Theta(n)$ time for the lower-order product bits $\langle p_{n-1}, p_{n-2}, \ldots, p_0 \rangle$ to be produced, and it takes $\Theta(n)$ time for the inputs to the adder to be ready. If the adder is a ripple-carry adder, it takes another $\Theta(n)$ time for the higher-order product bits $\langle p_{2n-1}, p_{2n-2}, \ldots, p_n \rangle$ to emerge. If the adder is a carry-lookahead adder, only $\Theta(\lg n)$ time is needed, but the total time remains $\Theta(n)$.

There are n^2 AND gates and $n^2 - n$ full adders in the array multiplier. The adder for the high-order output bits contributes only another $\Theta(n)$ gates. Thus, the array multiplier has $\Theta(n^2)$ size.

28.3.2 Wallace-tree multipliers

A ***Wallace tree*** is a circuit that reduces the problem of summing n n-bit numbers to the problem of summing two $\Theta(n)$-bit numbers. It does this by using $\lfloor n/3 \rfloor$ carry-save adders in parallel to convert the sum of n numbers to the sum of $\lceil 2n/3 \rceil$ numbers. It then recursively constructs a Wallace tree on the $\lceil 2n/3 \rceil$ resulting numbers. In this way, the set of numbers is progressively reduced until there are only two numbers left. By performing many carry-save additions in parallel, Wallace trees allow two n-bit numbers to be multiplied in $\Theta(\lg n)$ time using a circuit with $\Theta(n^2)$ size.

Figure 28.16 shows a Wallace tree[2] that adds 8 partial products $m^{(0)}, m^{(1)}, \ldots, m^{(7)}$. Partial product $m^{(i)}$ consists of $n + i$ bits. Each line represents an entire number, not just a single bit; next to each line is the number of bits the line represents (see Exercise 28.3-3). The carry-lookahead adder at the bottom adds a $(2n - 1)$-bit number to a $2n$-bit number to give the $2n$-bit product.

Analysis

The time required by an n-input Wallace tree depends on the depth of the carry-save adders. At each level of the tree, each group of 3 numbers is reduced to 2 numbers, with at most 2 numbers left over (as in the case of $m^{(6)}$ and $m^{(7)}$ at the top level). Thus, the maximum depth $D(n)$ of a carry-save adder in an n-input Wallace tree is given by the recurrence

$$D(n) = \begin{cases} 0 & \text{if } n \le 2 \,, \\ 1 & \text{if } n = 3 \,, \\ D(\lceil 2n/3 \rceil) + 1 & \text{if } n \ge 4 \,, \end{cases}$$

which has the solution $D(n) = \Theta(\lg n)$ by case 2 of the master theorem (Theorem 4.1). Each carry-save adder takes $\Theta(1)$ time. All n partial products can be formed in $\Theta(1)$ time in parallel. (The lowest-order $i - 1$ bits of $m^{(i)}$, for $i = 1, 2, \ldots, n - 1$, are hardwired to 0.) The carry-lookahead adder takes $O(\lg n)$ time. Thus, the entire multiplication of two n-bit numbers takes $\Theta(\lg n)$ time.

A Wallace-tree multiplier for two n-bit numbers has $\Theta(n^2)$ size, which we can see as follows. We first bound the circuit size of the carry-save adders. A lower

[2] As you can see from the figure, a Wallace tree is not truly a tree, but rather a directed acyclic graph. The name is historical.

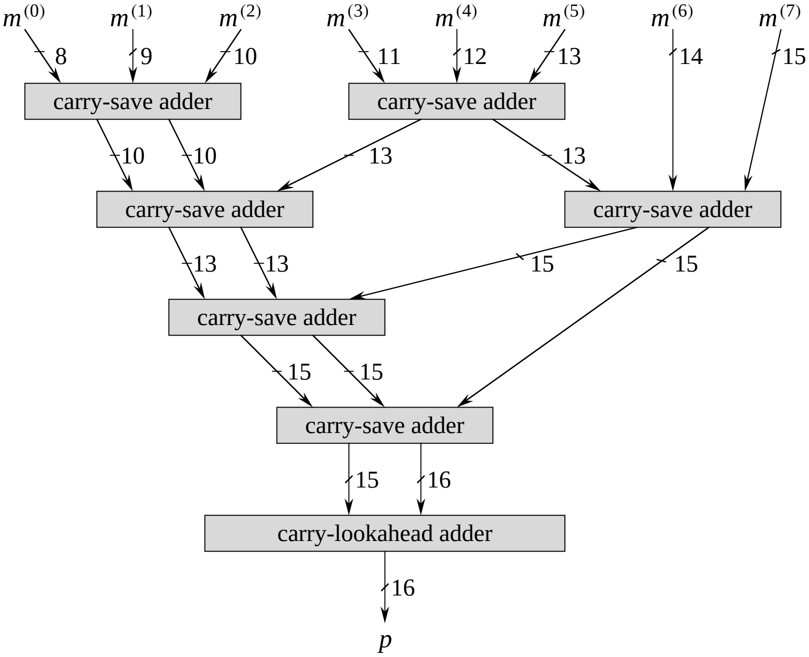


Figure 28.16 A Wallace tree that adds $n = 8$ partial products $m^{(0)}, m^{(1)}, \ldots, m^{(7)}$. Each line represents a number with the number of bits indicated. The left output of each carry-save adder represents the sum bits, and the right output represents the carry bits.

bound of $\Omega(n^2)$ is easy to obtain, since there are $\lfloor n/3 \rfloor$ carry-save adders at depth 1, and each one consists of at least n full adders. To get the upper bound of $O(n^2)$, observe that since the final product has $2n$ bits, each carry-save adder in the Wallace tree contains at most $2n$ full adders. We need to show that there are $O(n)$ carry-save adders altogether. Let $C(n)$ be the total number of carry-save adders in a Wallace tree with n input numbers. We have the recurrence

$$C(n) \le \begin{cases} 1 & \text{if } n = 3\,, \\ C(\lceil 2n/3 \rceil) + \lfloor n/3 \rfloor & \text{if } n \ge 4\,, \end{cases}$$

which has the solution $C(n) = \Theta(n)$ by case 3 of the master theorem. We thus obtain an asymptotically tight bound of $\Theta(n^2)$ size for the carry-save adders of a Wallace-tree multiplier. The circuitry to set up the n partial products has $\Theta(n^2)$ size, and the carry-lookahead adder at the end has $\Theta(n)$ size. Thus, the size of the entire multiplier is $\Theta(n^2)$.

Although the Wallace-tree-based multiplier is asymptotically faster than the array multiplier and has the same asymptotic size, its layout when it is implemented is not as regular as the array multiplier's, nor is it as "dense" (in the sense of having little wasted space between circuit elements). In practice, a compromise between the two designs is often used. The idea is to use two arrays in parallel, one adding up half of the partial products and one adding up the other half. The propagation delay is only half of that incurred by a single array adding up all n partial products. Two more carry-save additions reduce the 4 numbers output by the arrays to 2 numbers, and a carry-lookahead adder then adds the 2 numbers to yield the product. The total propagation delay is a little more than half that of a full array multiplier, plus an additional $O(\lg n)$ term.

Exercises

28.3-1

Prove that in an array multiplier, $v_j^{(i)} = 0$ for $i = 1, 2, \ldots, n-1$ and $j = 0, 1, \ldots, i, i+n, i+n+1, \ldots, 2n-1$.

28.3-2

Show that in the array multiplier of Figure 28.14, all but one of the full adders in the top row are unnecessary. You will need to do some rewiring.

28.3-3

Suppose that a carry-save adder takes inputs x, y, and z and produces outputs s and c, with n_x, n_y, n_z, n_s, and n_c bits respectively. Suppose also, without loss of generality, that $n_x \le n_y \le n_z$. Show that $n_s = n_z$ and that

$$n_c = \begin{cases} n_z & \text{if } n_y < n_z \ , \\ n_z + 1 & \text{if } n_y = n_z \ . \end{cases}$$

28.3-4

Describe an efficient circuit to compute the quotient when a binary number x is divided by 3. (*Hint:* Note that in binary, $.010101\ldots = .01 \times 1.01 \times 1.0001 \times \cdots$.)

28.3-5

A ***cyclic shifter***, or ***barrel shifter***, is a circuit that has two inputs $x = \langle x_{n-1}, x_{n-2}, \ldots, x_0 \rangle$ and $s = \langle s_{m-1}, s_{m-2}, \ldots, s_0 \rangle$, where $m = \lceil \lg n \rceil$. Its output $y = \langle y_{n-1}, y_{n-2}, \ldots, y_0 \rangle$ is specified by $y_i = x_{i+s \bmod n}$, for $i = 0, 1, \ldots, n-1$. That is, the shifter rotates the bits of x by the amount specified by s. Describe an efficient cyclic shifter. In terms of modular multiplication, what function does a cyclic shifter implement?

28.4 Clocked circuits

The elements of a combinational circuit are used only once during a computation. By introducing clocked memory elements into the circuit, we can reuse combinational elements. Because they can use hardware more than once, clocked circuits can often be much smaller than combinational circuits for the same function.

This section investigates clocked circuits for performing addition and multiplication. We begin with a $\Theta(1)$-size clocked circuit, called a bit-serial adder, that can add two n-bit numbers in $\Theta(n)$ time. We then investigate linear-array multipliers. We present a linear-array multiplier with $\Theta(n)$ size that can multiply two n-bit numbers in $\Theta(n)$ time.

28.4.1 Bit-serial addition

We introduce the notion of a clocked circuit by returning to the problem of adding two n-bit numbers. Figure 28.17 shows how we can use a single full adder to produce the $(n+1)$-bit sum $s = \langle s_n, s_{n-1}, \ldots, s_0 \rangle$ of two n-bit numbers $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ and $b = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$. The external world presents the input bits one pair at a time: first a_0 and b_0, then a_1 and b_1, and so forth. Although we want the carry-out from one bit position to be the carry-in to the next bit position, we cannot just feed the full adder's c output directly into an input. There is a timing issue: the carry-in c_i entering the full adder must correspond to the appropriate inputs a_i and b_i. Unless these input bits arrive at exactly the same moment as the fed-back carry, the output may be incorrect.

As Figure 28.17 shows, the solution is to use a ***clocked circuit***, or ***sequential circuit***, consisting of combinational circuitry and one or more ***registers*** (clocked memory elements). The combinational circuitry has inputs from the external world or from the output of registers. It provides outputs to the external world and to the input of registers. As in combinational circuits, we prohibit the combinational circuitry in a clocked circuit from containing cycles.

Each register in a clocked circuit is controlled by a periodic signal, or ***clock***. Whenever the clock pulses, or ***ticks***, the register loads in and stores the value at its input. The time between successive clock ticks is a ***clock period***. In a ***globally clocked*** circuit, every register works off the same clock.

Let us examine the operation of a register in a little more detail. We treat each clock tick as a momentary pulse. At a given tick, a register reads the input value presented to it *at that moment* and stores it. This stored value then appears at the register's output, where it can be used to compute values that are moved into other registers at the next clock tick. In other words, the value at a register's input during one clock period appears on the register's output during the next clock period.

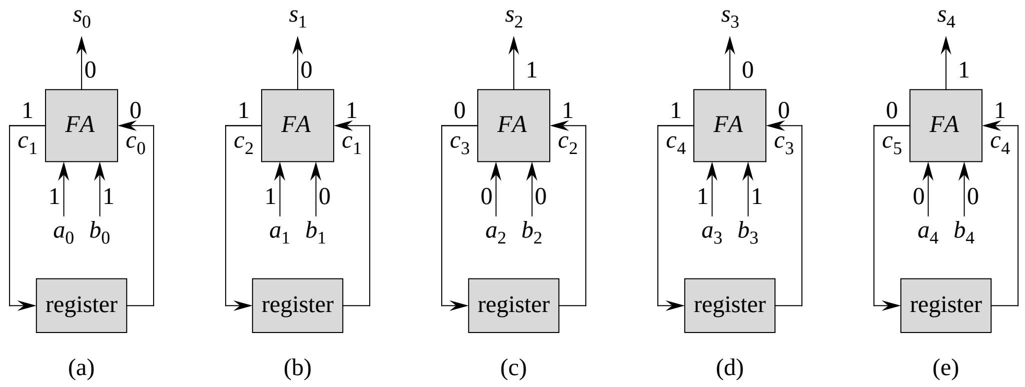


Figure 28.17 The operation of a bit-serial adder. During the ith clock period, for $i = 0, 1, \ldots, n$, the full adder *FA* takes input bits a_i and b_i from the outside world and a carry bit c_i from the register. The full adder then outputs sum bit s_i, which is provided externally, and carry bit c_{i+1}, which is stored back in the register to be used during the next clock period. The register is initialized with $c_0 = 0$. **(a)–(e)** The state of the circuit in each of the five clock periods during the addition of $a = \langle 1011 \rangle$ and $b = \langle 1001 \rangle$ to produce $s = \langle 10100 \rangle$.

Now let us examine the circuit in Figure 28.17, which we call a ***bit-serial adder***. In order for the full adder's outputs to be correct, we require that the clock period be at least as long as the propagation delay of the full adder, so that the combinational circuitry has an opportunity to settle between clock ticks. During clock period 0, shown in Figure 28.17(a), the external world applies input bits a_0 and b_0 to two of the full adder's inputs. We assume that the register is initialized to store a 0; the initial carry-in bit, which is the register output, is thus $c_0 = 0$. Later in this clock period, sum bit s_0 and carry-out c_1 emerge from the full adder. The sum bit goes to the external world, where presumably it will be saved as part of the entire sum s. The wire from the carry-out of the full adder feeds into the register, so that c_1 is read into the register upon the next clock tick. At the beginning of clock period 1, therefore, the register contains c_1. During clock period 1, shown in Figure 28.17(b), the outside world applies a_1 and b_1 to the full adder, which, reading c_1 from the register, produces outputs s_1 and c_2. The sum bit s_1 goes out to the outside world, and c_2 goes to the register. This cycle continues until clock period n, shown in Figure 28.17(e), in which the register contains c_n. The external world then applies $a_n = b_n = 0$, so that we get $s_n = c_n$.

Analysis

To determine the total time t taken by a globally clocked circuit, we need to know the number p of clock periods and the duration d of each clock period: $t = pd$.

The clock period d must be long enough for all combinational circuitry to settle between ticks. Although for some inputs it may settle earlier, if the circuit is to work correctly for all inputs, d must be at least proportional to the depth of the combinational circuitry.

Let us see how long it takes to add two n-bit numbers bit-serially. Each clock period takes $\Theta(1)$ time because the depth of the full adder is $\Theta(1)$. Since $n + 1$ clock ticks are required to produce all the outputs, the total time to perform bit-serial addition is $(n + 1)\,\Theta(1) = \Theta(n)$.

The size of the bit-serial adder (number of combinational elements plus number of registers) is $\Theta(1)$.

Ripple-carry addition versus bit-serial addition

Observe that a ripple-carry adder is like a replicated bit-serial adder with the registers replaced by direct connections between combinational elements. That is, the ripple-carry adder corresponds to a spatial "unrolling" of the computation of the bit-serial adder. The ith full adder in the ripple-carry adder implements the ith clock period of the bit-serial adder.

In general, we can replace any clocked circuit by an equivalent combinational circuit having the same asymptotic time delay if we know in advance how many clock periods the clocked circuit runs for. There is, of course, a trade-off involved. The clocked circuit uses fewer circuit elements (a factor of $\Theta(n)$ less for the bit-serial adder versus the ripple-carry adder), but the combinational circuit has the advantage of less control circuitry—we need no clock or synchronized external circuit to present input bits and store sum bits. Moreover, although the circuits have the same asymptotic time delay, the combinational circuit typically runs slightly faster in practice. The extra speed is possible because the combinational circuit doesn't have to wait for values to stabilize during each clock period. If all the inputs stabilize at once, values just ripple through the circuit at the maximum possible speed, without waiting for the clock.

28.4.2 Linear-array multipliers

The combinational multipliers of Section 28.3 need $\Theta(n^2)$ size to multiply two n-bit numbers. We now present two multipliers that are linear, rather than two-dimensional, arrays of circuit elements. Like the array multiplier, the faster of these two linear-array multipliers runs in $\Theta(n)$ time.

The linear-array multipliers implement the ***Russian peasant's algorithm*** (so called because Westerners visiting Russia in the nineteenth century found the algorithm widely used there), illustrated in Figure 28.18(a). Given two input numbers a and b, we make two columns of numbers, headed by a and b. In each row, the a-column entry is half of the previous row's a-column entry, with fractions dis-

a	b
19	29
9	58
4	116
2	232
1	464
	551

(a)

a	b
1 0 0 1 1	1 1 1 0 1
1 0 0 1	1 1 1 0 1 0
1 0 0	1 1 1 0 1 0 0
1 0	1 1 1 0 1 0 0 0
1	1 1 1 0 1 0 0 0 0
	1 0 0 0 1 0 0 1 1 1

(b)

Figure 28.18 Multiplying 19 by 29 with the Russian peasant's algorithm. The a-column entry in each row is half of the previous row's entry with fractions ignored, and the b-column entries double from row to row. We add the b-column entries in all rows with odd a-column entries, which are shaded. This sum is the desired product. **(a)** The numbers expressed in decimal. **(b)** The same numbers in binary.

carded. The b-column entry is twice the previous row's b-column entry. The last row is the one with an a-column entry of 1. We look at all the a-column entries that contain odd values and sum the corresponding b-column entries. This sum is the product $a \cdot b$.

Although the Russian peasant's algorithm may seem remarkable at first, Figure 28.18(b) shows that it is really just a binary-number-system implementation of the grade-school multiplication method, but with numbers expressed in decimal rather than binary. Rows in which the a-column entry is odd contribute to the product a term of b multiplied by the appropriate power of 2.

A slow linear-array implementation

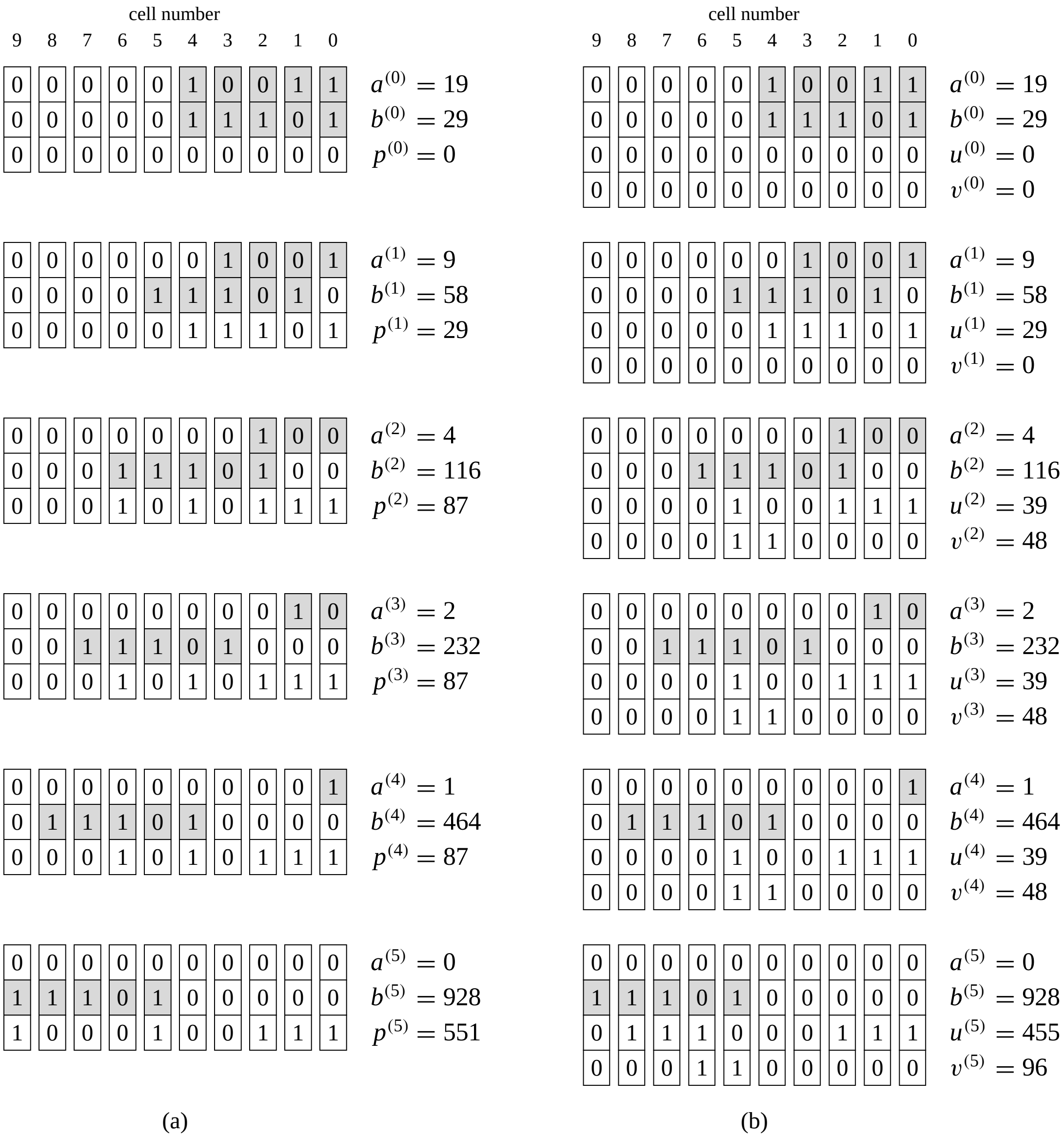
Figure 28.19(a) shows one way to implement the Russian peasant's algorithm for two n-bit numbers. We use a clocked circuit consisting of a linear array of $2n$ cells. Each cell contains three registers. One register holds a bit from an a entry, one holds a bit from a b entry, and one holds a bit of the product p. We use superscripts to denote cell values before each step of the algorithm. For example, the value of bit a_i before the jth step is $a_i^{(j)}$, and we define $a^{(j)} = \langle a_{2n-1}^{(j)}, a_{2n-2}^{(j)}, \ldots, a_0^{(j)} \rangle$.

The algorithm executes a sequence of n steps, numbered $0, 1, \ldots, n-1$, where each step takes one clock period. The algorithm maintains the following invariant:

Before the jth step,

$$a^{(j)} \cdot b^{(j)} + p^{(j)} = a \cdot b \; . \tag{28.6}$$

(See Exercise 28.4-2). Initially, $a^{(0)} = a$, $b^{(0)} = b$, and $p^{(0)} = 0$. The jth step consists of the following computations.



cell number

9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	1	0	0	1	1	$a^{(0)} = 19$
0	0	0	0	0	1	1	1	0	1	$b^{(0)} = 29$
0	0	0	0	0	0	0	0	0	0	$p^{(0)} = 0$
0	0	0	0	0	0	1	0	0	1	$a^{(1)} = 9$
0	0	0	0	1	1	1	0	1	0	$b^{(1)} = 58$
0	0	0	0	0	1	1	1	0	1	$p^{(1)} = 29$
0	0	0	0	0	0	0	1	0	0	$a^{(2)} = 4$
0	0	0	1	1	1	0	1	0	0	$b^{(2)} = 116$
0	0	0	1	0	1	0	1	1	1	$p^{(2)} = 87$
0	0	0	0	0	0	0	0	1	0	$a^{(3)} = 2$
0	0	1	1	1	0	1	0	0	0	$b^{(3)} = 232$
0	0	0	1	0	1	0	1	1	1	$p^{(3)} = 87$
0	0	0	0	0	0	0	0	0	1	$a^{(4)} = 1$
0	1	1	1	0	1	0	0	0	0	$b^{(4)} = 464$
0	0	0	1	0	1	0	1	1	1	$p^{(4)} = 87$
0	0	0	0	0	0	0	0	0	0	$a^{(5)} = 0$
1	1	1	0	1	0	0	0	0	0	$b^{(5)} = 928$
1	0	0	0	1	0	0	1	1	1	$p^{(5)} = 551$

(a)

cell number

9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	1	0	0	1	1	$a^{(0)} = 19$
0	0	0	0	0	1	1	1	0	1	$b^{(0)} = 29$
0	0	0	0	0	0	0	0	0	0	$u^{(0)} = 0$
0	0	0	0	0	0	0	0	0	0	$v^{(0)} = 0$
0	0	0	0	0	0	1	0	0	1	$a^{(1)} = 9$
0	0	0	0	1	1	1	0	1	0	$b^{(1)} = 58$
0	0	0	0	0	1	1	1	0	1	$u^{(1)} = 29$
0	0	0	0	0	0	0	0	0	0	$v^{(1)} = 0$
0	0	0	0	0	0	0	1	0	0	$a^{(2)} = 4$
0	0	0	1	1	1	0	1	0	0	$b^{(2)} = 116$
0	0	0	0	1	0	0	1	1	1	$u^{(2)} = 39$
0	0	0	0	1	1	0	0	0	0	$v^{(2)} = 48$
0	0	0	0	0	0	0	0	1	0	$a^{(3)} = 2$
0	0	1	1	1	0	1	0	0	0	$b^{(3)} = 232$
0	0	0	0	1	0	0	1	1	1	$u^{(3)} = 39$
0	0	0	0	1	1	0	0	0	0	$v^{(3)} = 48$
0	0	0	0	0	0	0	0	0	1	$a^{(4)} = 1$
0	1	1	1	0	1	0	0	0	0	$b^{(4)} = 464$
0	0	0	0	1	0	0	1	1	1	$u^{(4)} = 39$
0	0	0	0	1	1	0	0	0	0	$v^{(4)} = 48$
0	0	0	0	0	0	0	0	0	0	$a^{(5)} = 0$
1	1	1	0	1	0	0	0	0	0	$b^{(5)} = 928$
0	1	1	1	0	0	0	1	1	1	$u^{(5)} = 455$
0	0	0	1	1	0	0	0	0	0	$v^{(5)} = 96$

(b)

Figure 28.19 Two linear-array implementations of the Russian peasant's algorithm, showing the multiplication of $a = 19 = \langle 10011 \rangle$ by $b = 29 = \langle 11101 \rangle$, with $n = 5$. The situation at the beginning of each step j is shown, with the remaining significant bits of $a^{(j)}$ and $b^{(j)}$ shaded. **(a)** A slow implementation that runs in $\Theta(n^2)$ time. Because $a^{(5)} = 0$, we have $p^{(5)} = a \cdot b$. There are n steps, and each step uses a ripple-carry addition. The clock period is therefore proportional to the length of the array, or $\Theta(n)$, leading to $\Theta(n^2)$ time overall. **(b)** A fast implementation that runs in $\Theta(n)$ time because each step uses carry-save addition rather than ripple-carry addition, thus taking only $\Theta(1)$ time. There are a total of $2n - 1 = 9$ steps; after the last step shown, repeated carry-save addition of u and v yields $u^{(9)} = a \cdot b$.

1. If $a^{(j)}$ is odd (that is, $a_0^{(j)} = 1$), then add b into p: $p^{(j+1)} \leftarrow b^{(j)} + p^{(j)}$. (The addition is performed by a ripple-carry adder that runs the length of the array; carry bits ripple from right to left.) If $a^{(j)}$ is even, then carry p through to the next step: $p^{(j+1)} \leftarrow p^{(j)}$.
2. Shift a right by one bit position:
$$a_i^{(j+1)} \leftarrow \begin{cases} a_{i+1}^{(j)} & \text{if } 0 \le i \le 2n-2 \,, \\ 0 & \text{if } i = 2n-1 \,. \end{cases}$$
3. Shift b left by one bit position:
$$b_i^{(j+1)} \leftarrow \begin{cases} b_{i-1}^{(j)} & \text{if } 1 \le i \le 2n-1 \,, \\ 0 & \text{if } i = 0 \,. \end{cases}$$

After running n steps, we have shifted out all the bits of a; thus, $a^{(n)} = 0$. Invariant (28.6) then implies that $p^{(n)} = a \cdot b$.

We now analyze the algorithm. There are n steps, assuming that the control information is broadcast to each cell simultaneously. Each step takes $\Theta(n)$ time in the worst case, because the depth of the ripple-carry adder is $\Theta(n)$, and thus the duration of the clock period must be at least $\Theta(n)$. Each shift takes only $\Theta(1)$ time. Overall, therefore, the algorithm takes $\Theta(n^2)$ time. Because each cell has constant size, the entire linear array has $\Theta(n)$ size.

A fast linear-array implementation

By using carry-save addition instead of ripple-carry addition, we can decrease the time for each step to $\Theta(1)$, thus improving the overall time to $\Theta(n)$. As Figure 28.19(b) shows, once again each cell contains a bit of an a entry and a bit of a b entry. Each cell also contains two more bits, from u and v, which are the outputs from carry-save addition. Using a carry-save representation to accumulate the product, we maintain the following invariant:

Before the jth step,

$$a^{(j)} \cdot b^{(j)} + u^{(j)} + v^{(j)} = a \cdot b \,. \qquad (28.7)$$

(Again, see Exercise 28.4-2). Each step shifts a and b in the same way as the slow implementation, so that we can combine equations (28.6) and (28.7) to yield $u^{(j)} + v^{(j)} = p^{(j)}$. Thus, the u and v bits contain the same information as the p bits in the slow implementation.

The jth step of the fast implementation performs carry-save addition on u and v, where the operands depend on whether a is odd or even. If $a_0^{(j)} = 1$, we compute

$$u_i^{(j+1)} \leftarrow \text{parity}(b_i^{(j)}, u_i^{(j)}, v_i^{(j)}) \qquad \text{for } i = 0, 1, \ldots, 2n-1$$

and

$$v_i^{(j+1)} \leftarrow \begin{cases} \text{majority}(b_{i-1}^{(j)}, u_{i-1}^{(j)}, v_{i-1}^{(j)}) & \text{if } 1 \le i \le 2n-1 \,, \\ 0 & \text{if } i = 0 \,. \end{cases}$$

Otherwise, $a_0^{(j)} = 0$, and we compute

$$u_i^{(j+1)} \leftarrow \text{parity}(0, u_i^{(j)}, v_i^{(j)}) \qquad \text{for } i = 0, 1, \ldots, 2n-1$$

and

$$v_i^{(j+1)} \leftarrow \begin{cases} \text{majority}(0, u_{i-1}^{(j)}, v_{i-1}^{(j)}) & \text{if } 1 \le i \le 2n-1 \,, \\ 0 & \text{if } i = 0 \,. \end{cases}$$

After updating u and v, the jth step shifts a to the right and b to the left in the same manner as the slow implementation.

The fast implementation performs a total of $2n - 1$ steps. For $j \ge n$, we have $a^{(j)} = 0$, and invariant (28.7) therefore implies that $u^{(j)} + v^{(j)} = a \cdot b$. Once $a^{(j)} = 0$, all further steps serve only to carry-save add u and v. Exercise 28.4-3 asks you to show that $v^{(2n-1)} = 0$, so that $u^{(2n-1)} = a \cdot b$.

The total time in the worst case is $\Theta(n)$, since each of the $2n - 1$ steps takes $\Theta(1)$ time. Because each cell still has constant size, the total size remains $\Theta(n)$.

Exercises

28.4-1

Let $a = \langle 101101 \rangle$, $b = \langle 011110 \rangle$, and $n = 6$. Show how the Russian peasant's algorithm operates, in both decimal and binary, for inputs a and b.

28.4-2

Prove that the invariants (28.6) and (28.7) for the linear-array multipliers are hold initially and that they are maintained by each step.

28.4-3

Prove that in the fast linear-array multiplier, $v^{(2n-1)} = 0$.

28.4-4

Describe how the array multiplier from Section 28.3.1 represents an "unrolling" of the computation of the fast linear-array multiplier.

28.4-5

Consider a data stream $\langle x_1, x_2, \ldots \rangle$ that arrives at a clocked circuit at the rate of 1 value per clock tick. For a fixed value n, the circuit must compute the value

$$y_t = \max_{t-n+1 \le i \le t} x_i$$

for $t = n, n+1, \ldots$. That is, y_t is the maximum of the most recent n values received by the circuit. Give an $O(n)$-size circuit that on each clock tick inputs the value x_t and computes the output value y_t in $O(1)$ time. The circuit can use registers and combinational elements that compute the maximum of two inputs.

28.4-6 ⋆

Redo Exercise 28.4-5 using only $O(\lg n)$ "maximum" elements.

Problems

28-1 Division circuits

We can construct a division circuit from subtraction and multiplication circuits with a technique called ***Newton iteration***. We shall focus on the related problem of computing a reciprocal, since we can obtain a division circuit by making one additional multiplication.

The idea is to compute a sequence $y_0, y_1, y_2, \ldots$ of approximations to the reciprocal of a number x by using the formula

$$y_{i+1} \leftarrow 2y_i - xy_i^2 .$$

Assume that x is given as an n-bit binary fraction in the range $1/2 \le x \le 1$. Since the reciprocal can be an infinite repeating fraction, we shall concentrate on computing an n-bit approximation accurate up to its least significant bit.

a. Suppose that $|y_i - 1/x| \le \epsilon$ for some constant $\epsilon > 0$. Prove that $|y_{i+1} - 1/x| \le \epsilon^2$.

b. Give an initial approximation y_0 such that y_k satisfies $|y_k - 1/x| \le 2^{-2^k}$ for all $k \ge 0$. How large must k be for the approximation y_k to be accurate up to its least significant bit?

c. Describe a combinational circuit that, given an n-bit input x, computes an n-bit approximation to $1/x$ in $O(\lg^2 n)$ time. What is the size of your circuit? (*Hint:* With a little cleverness, you can beat the size bound of $\Theta(n^2 \lg n)$.)

28-2 Boolean formulas for symmetric functions

A n-input function $f(x_1, x_2, \ldots, x_n)$ is ***symmetric*** if

$$f(x_1, x_2, \ldots, x_n) = f(x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)})$$

for any permutation π of $\{1, 2, \ldots, n\}$. In this problem, we shall show that there is a boolean formula representing f whose size is polynomial in n. (For our purposes, a boolean formula is a string comprised of the variables $x_1, x_2, \ldots, x_n$, parentheses, and the boolean operators $\vee$, $\wedge$, and $\neg$.) Our approach will be to convert a logarithmic-depth boolean circuit to an equivalent polynomial-size boolean formula. We shall assume that all circuits are constructed from 2-input AND, 2-input OR, and NOT gates.

a. We start by considering a simple symmetric function. The generalized ***majority function*** on n boolean inputs is defined by

$$\text{majority}_n(x_1, x_2, \ldots, x_n) = \begin{cases} 1 & \text{if } x_1 + x_2 + \cdots + x_n > n/2 \ , \\ 0 & \text{otherwise} \ . \end{cases}$$

Describe an $O(\lg n)$-depth combinational circuit for majority_n. (*Hint:* Build a tree of adders.)

b. Suppose that f is an arbitrary boolean function of the n boolean variables $x_1, x_2, \ldots, x_n$. Suppose further that there is a circuit C of depth d that computes f. Show how to construct from C a boolean formula for f of length $O(2^d)$. Conclude that there is polynomial-size formula for majority_n.

c. Argue that any symmetric boolean function $f(x_1, x_2, \ldots, x_n)$ can be expressed as a function of $\sum_{i=1}^{n} x_i$.

d. Argue that any symmetric function on n boolean inputs can be computed by an $O(\lg n)$-depth combinational circuit.

e. Argue that any symmetric boolean function on n boolean variables can be represented by a boolean formula whose length is polynomial in n.

Chapter notes

Most books on computer arithmetic focus more on practical implementations of circuitry than on algorithmic theory. Savage [274] is one of the few that investigates algorithmic aspects of the subject. The more hardware-oriented books on computer arithmetic by Cavanagh [56] and Hwang [170] are especially readable. Good books on combinational and sequential logic design include Hill and Peterson [151] and, with a twist toward formal language theory, Kohavi [195].

Aiken and Hopper [11] trace the early history of arithmetic algorithms. Ripple-carry addition is as at least as old as the abacus, which has been around for over

5000 years. The first mechanical calculator employing ripple-carry addition was devised by B. Pascal in 1642. A calculating machine adapted to repeated addition for multiplication was conceived by S. Morland in 1666 and independently by G. W. Leibniz in 1671. The Russian peasant's algorithm for multiplication is apparently much older than its use in Russia in the nineteenth century. According to Knuth [190], it was used by Egyptian mathematicians as long ago as 1800 B.C.

The kill, generate, and propagate statuses of a carry chain were exploited in a relay calculator built at Harvard during the mid-1940's [295]. One of the first implementations of carry-lookahead addition was described by Weinberger and Smith [322], but their lookahead method requires large gates. Ofman [240] proved that n-bit numbers could be added in $O(\lg n)$ time using carry-lookahead addition with constant-size gates.

The idea of using carry-save addition to speed up multiplication is due to Estrin, Gilchrist, and Pomerene [91]. Atrubin [26] describes a linear-array multiplier of infinite length that can be used to multiply binary numbers of arbitrary length. The multiplier produces the nth bit of the product immediately upon receiving the nth bits of the inputs. The Wallace-tree multiplier is attributed to Wallace [319], but the idea was also independently discovered by Ofman [240].

Division algorithms date back to I. Newton, who around 1665 invented what has become known as Newton iteration. Problem 28-1 uses Newton iteration to construct a division circuit with $\Theta(\lg^2 n)$ depth. This method was improved by Beame, Cook, and Hoover [35], who showed that n-bit division can in fact be performed in $\Theta(\lg n)$ depth.