The complexity of computations by networks

by Nicholas Pippenger

We survey the current state of knowledge concerning the computation of Boolean functions by networks, with particular emphasis on the addition and multiplication of binary numbers.

1. Introduction

The object of this paper is to survey what is known about the complexity of computing certain Boolean functions. The model of computation we use is that of Boolean networks, also known as combinational logic networks or circuits. The functions we deal with are those corresponding to arithmetic operations on numbers represented in binary. These functions are among the most important of those that networks are commonly used to compute; they also have the merit of illustrating nicely much of the theory of computation by networks.

Though the computation of arithmetic functions by networks is one of considerable practical importance, the viewpoint of this paper is ruthlessly theoretical. Most textbooks on computer arithmetic devote a great deal of discussion to matters such as the representation of negative numbers. Though this issue has practical importance, it is of little theoretical interest, and almost nothing is said about it here. Many of the results we describe, on the other hand, have practical relevance only if it is necessary to perform operations on numbers of astronomical length; they are unlikely to find application outside of stunts such as the computation of π to millions of digits. These results are of theoretical importance, however, because they describe the fundamental possibilities and limitations of computation by networks.

We confine our attention to operations on numbers represented in binary. Little would change if we adopted decimal, or any other fixed radix, instead. Because we wish to study the complexity of computing prescribed Boolean functions, we regard the representation system as given. One may, however, regard the representation system as something to be engineered, like the network; for results from this point of view, see the pioneering work of Winograd [1, 2].

Let **B** denote the Boolean algebra with two elements, which are denoted 0 and 1 (alternative denotations are "false" and "true," respectively). A Boolean function f depending on n arguments is simply a map $f{:}\mathbf{B}^n \rightarrow \mathbf{B}$. Suppose that we are given the values $x_1, \cdots, x_n$ of the arguments and that we wish to compute the value $f(x_1, \cdots, x_n)$. In this situation we may regard $x_1, \cdots, x_n$ as indeterminates and f as an element of the extension Boolean algebra $\mathbf{B}(x_1, \cdots, x_n)$, which contains 2^{2^n} elements. (Recall the analogous situation in which a real polynomial in n variables may be regarded either as a map $p{:}\mathbf{R}^n \rightarrow \mathbf{R}$ or an element of the ring $\mathbf{R}[x_1, \cdots, x_n]$.)

The idea of computation by Boolean networks is a simple one: We are given a supply of components called "gates" that compute some basic Boolean functions, and we wish to interconnect them into a system called a "network" that computes one or more other Boolean functions. We illustrate this idea with an example. Suppose we are given a supply of gates that compute the "nand" function of two arguments (this function assumes the value 1 unless both its arguments assume the value 1, in which case it assumes the value 0). Suppose that we wish to compute the "parity" function of two arguments (also known as the "sum modulo 2"; it assumes the value 1 if and only if an odd number of its arguments assume the value 1). We can perform our task as follows.

$a \leftarrow \mathrm{nand}(x, y)$,

$b \leftarrow \mathrm{nand}(x, a)$,

$c \leftarrow \mathrm{nand}(a, y)$,

$z \leftarrow \mathrm{nand}(b, c)$.

In this example, x and y represent the network inputs, a, b, and c represent "wires" carrying the outputs of the first three gates to succeeding gates, and z represents the network output. The example illustrates several properties of networks. First, it is possible to compute something once and use it many times (the value of a is used twice, as are the network inputs x and y). Second, there are no cycles of dependence; each computation uses only network inputs or outputs of preceding computations.

Two important parameters of a network are its *size* (the number of gates it contains; four in the example) and its *depth* (the number of gates on the longest path of dependence; three in the example). The size of a network has an obvious relation to its cost (under the simplifying assumption that all gates have equal cost and that no other components, such as wires, have any cost). The depth corresponds (under analogous simplifying assumptions) to the delay introduced by a network.

We assume that we are given a set of available gate functions and a set of desired network functions, and we are interested in the minimum possible size of a network performing this task. Alternatively, we may be interested in the minimum possible depth, or in the possible combinations of size and depth that can be achieved simultaneously.

More generally, we may assume that we are given not only a set of available gate functions, but also an assignment of a nonnegative real *cost* and *delay* to each of them. Such a set of functions together with costs and delays will be called a *basis*. To any network built from these gates we also assign a cost (the sum of the costs of its constituent gates) and a delay (the maximum over all paths from an input to an output of the sum of the delays of the gates on that path). Given a set of desired network functions, we are interested in the minimum possible cost, or the minimum possible delay, or the achievable combinations of both.

It may seem at first that each possible basis gives rise to its own distinct complexity theory. This is true to a certain extent, but there are large classes of bases that behave, for theoretical purposes, in the same way. Consider, for example, bases having the following three properties. First, they contain only finitely many types of gates. Second, the gates they contain can be interconnected in sufficient quantities to form a network computing any prescribed Boolean function. Third, they contain gates computing the constant functions 0 and 1 with cost 0 and delay 0, but no nonconstant function has cost 0 or delay 0. Such a basis will be called a *standard bounded basis.*

Standard bounded bases have a number of pleasant properties that make their complexity theory particularly simple and natural. The first of these is that optimal costs and delays do not depend very strongly on which standard bounded basis we consider. Given any two standard bounded bases, we can construct for each gate of one a network computing the same function using gates of the other. These networks can be connected together in the same way as gates, so in going from one standard bounded basis to another we need increase the cost and delay by at most constant factors (which depend only on the two bases). We celebrate this fact by stating results in a form that ignores constant factors: We usually give upper bounds in the form $O(g(n))$, meaning "bounded above by some constant times $g(n)$," and lower bounds in the form $\Omega(g(n))$, meaning "bounded below by some strictly positive constant times $g(n)$." This fact, which gives great coherence to the complexity theory of standard bounded bases, seems to have been appreciated by the earliest workers in the field (see Muller [3]).

The second pleasant property of standard bounded bases is that the cost and delay of a network computing a given Boolean function can be bounded above by simple functions of the number of arguments on which the function depends. Specifically, a function of n arguments is computed by a network having cost $O(2^n)$ and delay $O(n)$, simultaneously. To see this, assume (by virtue of the preceding pleasant property) that the basis contains a gate depending on three arguments (say, x, y, and z) and producing the output "if x then y else z." A network computing $f(x_1, \cdots, x_n)$ can then be constructed from such a gate with x_1 for x, the output of a network computing $f(1, x_2, \cdots, x_n)$ for y and the output of a network computing $f(0, x_2, \cdots, x_n)$ for z. It is easy to see that this construction yields the bounds mentioned above. The cost bound can be improved to $O(2^n/n)$ (see [3]), but this improvement is best possible, as is the delay bound.

The third pleasant property is that cost and delay are bounded below by simple functions of the number of arguments on which a function depends. (Here we must assume of course that the function actually depends on all its arguments, that is, that the two partial functions obtained by substituting the constants 0 and 1 for an argument are always distinct; otherwise, the function might be a constant in disguise.) For any standard bounded basis there is a maximum number, say k, of arguments on which any gate depends. It is easy to see by induction that if a network has size c, it can depend on at most $c(k - 1) + 1$ inputs, and if it has depth d, it can depend on at most k^d inputs. From this it follows that a network computing a function depending on n arguments must have cost $\Omega(n)$ and delay $\Omega(\log n)$. These bounds also seem to have been appreciated by early workers. Because they differ exponentially from the universally applicable upper bounds, they provide great scope within which functions can vary in their complexity.

A fourth pleasant property concerns an apparent asymmetry between inputs and outputs in networks. For a standard bounded basis, there is a bound to the number of arguments on which any gate depends (this number is sometimes called the "fan-in" of the gate). Suppose that we also postulate for each type of gate in the basis a bound to the number of other gates that can depend directly upon it (such a bound is called a "fan-out" bound for the gate). How would this affect optimal costs and delays? According to a theorem of Hoover, Klawe, and Pippenger [4], as long as the basis contains a nonconstant gate with a fan-out bound at least 2, both cost and delay are within constant factors of what they would be with unbounded fan-out. (To prove this for cost alone or delay alone is straightforward, but some care is needed to keep both under control simultaneously.)

These four pleasant properties, especially the first, provide strong justification for the complexity theory of standard bounded bases as "the" complexity theory of Boolean functions. A great frustration of this theory, however, is the paucity of results concerning specific functions. Although it is known that there exist functions of n arguments with costs almost everywhere from $O(2^n/n)$ down to $O(n)$ and delays almost everywhere from $O(n)$ down to $O(\log n)$ (see Paterson and Wegener [5], for example), no specific function is known for which the cost can be proved to be $O(n^2)$ but not $O(n)$, or for which the delay can be proved to be $O((\log n)^2)$ but not $O(\log n)$.

This frustration has motivated the study of "monotone" Boolean functions by monotone Boolean networks. A *monotone* Boolean function is one that is nondecreasing in the usual sense: Changing an argument from 0 to 1 can change the value from 0 to 1 but not from 1 to 0. The property of being monotone is preserved by composition, so if the gates in a basis compute only monotone functions, so will all networks over that basis. We may define a *monotone bounded basis* by altering the second condition in the definition of a standard bounded basis to require only computing any prescribed monotone Boolean function. Monotone bounded bases enjoy the four pleasant properties mentioned above (with the best possible universally applicable cost bound reduced to $O(2^n/n^{3/2})$; see Andreev [6]). There has been considerable success in proving lower bounds for the computation of sets of monotone Boolean functions by networks over monotone bounded bases, and recently there has been decisive progress in proving such bounds for single monotone Boolean functions (see Razborov [7] and Andreev [8] and the references cited therein). The topic of monotone computation is somewhat off the track of this paper, however, since most functions of arithmetic interest are not monotone.

Apart from standard bounded bases, the basis that concerns us most in this paper is one we call the *standard unbounded basis.* It comprises (1) a gate computing the negation of one argument with cost 1 and delay 0, (2) for each $k \geq 2$, a gate computing the "and" of k arguments with cost k and delay 1, and (3) for each $k \geq 2$, a gate computing the "or" of k arguments with cost k and delay 1. It is not hard to see that, as far as cost is concerned, the complexity theory for this basis is within constant factors the same as that of the standard bounded bases. Where delay is concerned, however, the standard unbounded basis has dramatically different properties from the standard bounded bases. To see this, let us consider how the pleasant properties of standard bounded bases are affected by the change.

First, consider the effect of changes to the basis. If we omit (1), we get a basis, the *monotone unbounded basis*, that bears the same relationship to the standard unbounded basis that the monotone bounded bases do to the standard bounded bases. Alternatively, we could by using DeMorgan's laws omit either (2) or (3) and obtain a basis differing by at most constant factors in the cost and differing not at all in delay. We see later, however, that adding other unbounded families of gates (for each $k \geq 2$, a gate computing the parity of k arguments with cost k and delay 1, for example) greatly affects the properties of the basis.

For a universally applicable upper bound, a function of n arguments is computed by a network of cost at most $n2^n$ and delay at most 2. To see this, express the function as the "or" of at most 2^n functions, each corresponding to one possible assignment of 0s and 1s to the arguments and each computed by the "and" of n arguments or their negations. We have a universally applicable lower bound of n for the cost of a network computing a function depending on n arguments, but as we have just seen, no function requires delay greater than 2.

Networks over the standard unbounded basis are often called "networks with unbounded fan-in," and it is invariably assumed that they also have unbounded fan-out. Similarly, networks over the standard bounded basis are often called "networks with bounded fan-in," and we have seen that it does not matter whether or not they have bounded fan-out.

2. Addition in binary

Suppose that we are given two natural numbers x and y, each in the range $\{0, \cdots, 2^n - 1\}$, as represented by n-bit binary sequences $x_0, \cdots, x_{n-1}$ and $y_0, \cdots, y_{n-1}$:

$$x = \sum_{0 \leq m \leq n-1} x_m 2^m$$

and

$$y = \sum_{0 \leq m \leq n-1} y_m 2^m.$$

We wish to compute their sum $z = x + y$, which lies in the range $\{0, \cdots, 2^{n+1} - 1\}$, as represented by an $(n + 1)$-bit binary sequence $z_0, \cdots, z_n$:

$$z = \sum_{0 \le m \le n} z_m 2^m.$$

Since z_n depends on all $2n$ inputs, it is clear that a network with bounded fan-in for addition must have cost $\Omega(n)$ and delay $\Omega(\log n)$. Our first goal in this section is to see how these bounds can be achieved.

A little thought reveals that the essence of the matter is the computation of the "carries" $c_0, \cdots, c_n$, which may be accomplished by means of the recurrence

$$c_{m+1} = \text{majority}\,(x_m, y_m, c_m)$$

for $0 \le m \le n - 1$, together with the condition $c_0 = 0$. (The "majority" function of n arguments assumes the value 1 if and only if at least half the arguments assume the value 1.) Once $c_0, \cdots, c_n$ have been computed, $z_0, \cdots, z_n$ can be computed by the formula

$$z_m = \text{parity}\,(x_m, y_m, c_m)$$

for $0 \le m \le n - 1$, together with $z_n = c_n$. These formulae yield a network with bounded fan-in for addition, often called a "ripple-carry adder," having cost $O(n)$ and delay $O(n)$. This settles the question of the optimal cost, but leaves open that of the optimal delay.

In 1956, Nadler [9], and independently Weinberger and Smith [10], showed how addition can be performed with delay $O(\log n)$. The networks they proposed (Nadler's is called a "pyramidal adder," Weinberger and Smith's a "carry-look-ahead adder") have cost $O(n \log n)$ rather than $O(n)$, as does yet another (called a "conditional-sum adder") proposed by Sklansky [11] in 1960. Thus, while these results settled the question of the optimal delay, they left open that of whether the optimal cost and delay can be achieved simultaneously. It was Ofman [12], in 1962, who solved this problem by showing how to achieve linear cost and logarithmic delay simultaneously.

Unlike its predecessors, Ofman's adder never acquired a catchy name. The idea behind it, however, has proved useful for a variety of other problems, and is now known as "parallel-prefix computation."

To describe Ofman's result, let us consider a finite-state automaton that receives a sequence of input symbols $a_0, \cdots, a_{n-1}$. When it receives the symbol a_m, it uses this symbol and its current state q_m to compute its next state q_{m+1} and an output symbol b_m. The initial state q_0 is assumed to be fixed. Ofman's result is that the computation of the output sequence $b_0, \cdots, b_{n-1}$ and the final state q_n from the input sequence $a_0, \cdots, a_{n-1}$ can be performed by a network with bounded fan-in having linear cost and logarithmic delay. (It does not matter how the input and output sequences are represented by Boolean variables, as long as each symbol is represented separately and in the same way; any desired change of representation can be accomplished by networks with linear cost and bounded delay, which does not affect our results.)

To apply this result to addition, we take the input symbol a_m to be the pair (x_m, y_m) of input bits (so there are four input symbols) and the state q_m to be the carry c_m (there are two states). The output bit z_m is then the output symbol b_m if $0 \le m \le n - 1$, or the final state q_n if $m = n$. The reader may enjoy the exercise of applying the result to the problem of dividing an n-bit binary number by 3, producing an $(n - 1)$-bit quotient and a 2-bit remainder.

To prove Ofman's result, we begin by regarding each input symbol a_m as a map A_m from the set of states to itself sending q_m to q_{m+1}. The effect of a sequence of symbols $a_l, \cdots, a_m$ is then given by the composition of maps $A_m \circ \cdots \circ A_l$. Clearly it suffices to compute some representation of the sequence of maps $Q_1 = A_0$, $Q_2 = A_1 \circ A_0, \cdots, Q_n = A_{n-1} \circ \cdots \circ A_0$, for then with a network of linear cost and bounded delay we may apply them to the initial state q_0 to get the sequence of states $q_1, \cdots, q_n$, then use these and the input sequence to compute the output sequence $b_0, \cdots, b_{n-1}$.

To see how to compute $Q_1, \cdots, Q_n$ from $A_0, \cdots, A_{n-1}$, suppose that $n = 2n'$ is even (the case of n odd is similar). First, compute the compositions $A'_m = A_{2m+1} \circ A_{1m}$ for $0 \le m \le n'$. Then, by recursive application of the procedure being described, compute $Q'_1 = A'_0$, $Q'_2 = A'_1 \circ A'_0, \cdots, Q'_{n'} = A'_{n'-1} \circ \cdots \circ A'_0$. Finally, compute $Q_1 = A_0$, $Q_{2m} = Q'_m$ for $1 \le m \le n'$ and $Q_{2m+1} = A_{2m} \circ Q'_m$ for $0 \le m \le n' - 1$. Since this recursion reduces n by a factor of two with linear cost and bounded delay, it yields a network with linear cost and logarithmic delay, which completes the proof of Ofman's result.

The only property of the composition of maps used in the foregoing construction is associativity, so the result is actually one about computation in semigroups, and this is the natural setting for parallel-prefix computation. Associated with the addition problem is a semigroup, the "carry semigroup," which contains three elements. Viewed as maps from the set of states to itself, these elements are the constant maps 0 and 1, and the identity map. In the context of addition, they correspond to "absorption" of a carry [accomplished by the input symbol (0, 0)], "generation" of a carry [accomplished by (1, 1)], and "propagation" of a carry [accomplished by (0, 1) or (1, 0)]. Below we meet other finite semigroups of computational interest.

Refinements of Ofman's work on addition have been given by Khrapchenko [13]. Further work on parallel-prefix computations has been done by Ladner and Fischer [14], Fich [15], and Snir [16].

Let us now consider addition by networks with unbounded fan-in. Adders with linear cost are still optimal (as regards cost), but now we should strive for bounded delay rather than logarithmic delay. As we saw in the introduction, we can achieve delay 2 with cost $O(n2^{2n})$. Thus the question is "Can bounded delay and linear cost be achieved simultaneously?"

As was the case for bounded fan-in, addition can be reduced to carry computation by a network with linear cost and bounded delay. Furthermore, the reverse reduction can be accomplished by a network with the same cost and delay, since

$$c_m = \text{parity}\,(x_m, y_m, z_m)$$

for $0 \le m \le n-1$ and $c_n = z_n$. Thus the question as to whether addition can be performed by networks with linear cost and bounded delay is equivalent to that as to whether carry computation can be performed by such networks. The latter question is somewhat more convenient to deal with, however. In particular, the carries are monotone functions of the inputs (since the "majority" function is monotone), so the question can be asked in two forms: for monotone networks with unbounded fan-in and for nonmonotone networks with unbounded fan-in.

It is implicit in the work of Weinberger and Smith [10] that the carries can be computed by a monotone network of delay 3 and cost $O(n^3)$. The next step was taken by Chandra, Fortune, and Lipton [17], who showed that carries can be computed by a monotone network of delay 6 and cost $O(n(\log n)^2)$. They went further than this, however; to describe their full result we must introduce some notation. For $n \ge 1$, let

$$\log^* n = \min\{l \ge 0 : \underbrace{\log \cdots \log}_{l} n \le 1\},$$

where all logs are to base 2. The values of log* at 1, 2, 4, 16, and 65 536 are 0, 1, 2, 3, and 4, respectively, so log* is a very slowly growing function. Let

$$\log^{**} n = \min\{l \ge 0 : \underbrace{\log^* \cdots \log^*}_{l} n \le 1\},$$

and, more generally, let

$$\log^{\overbrace{* \cdot \cdot \cdot *}^{k}} n = \min\{l \ge 0 : \underbrace{\log^{\overbrace{* \cdot \cdot \cdot *}^{k-1}} \cdots \log^{\overbrace{* \cdot \cdot \cdot *}^{k-1}}}_{l} n\}.$$

Chandra, Fortune, and Lipton showed that the carries can be computed by a monotone network of delay $6k$ and cost

$$O(n(\log^{\overbrace{* \cdot \cdot \cdot *}^{k-1}} n)^2).$$

This result says that one can almost, though not quite, construct adders of bounded delay and linear cost. (In view of the astronomical argument values needed for functions such as log** to assume large values, it is clear that this result is of a purely theoretical character!)

Like the result of Ofman, the result of Chandra, Fortune, and Lipton can be extended to other finite semigroups. It extends to any finite semigroup if gates are available that compute the product of k semigroup elements with cost k and delay 1. If only gates in the standard unbounded basis are available, it extends to those finite semigroups that contain no group as a subsemigroup. (Semigroups that do contain a group are discussed in the next section.)

We are still left with the question of whether there are adders with bounded delay and linear cost. The first step in answering this question was also taken by Chandra, Fortune, and Lipton [18], who showed that any monotone network for computing the carries with delay k must have cost at least

$$\Omega(n \log^{\overbrace{* \cdot \cdot \cdot *}^{k-2}} n).$$

They did this by showing that any monotone network for computing the carries has to be what they call a "parallel-prefix graph," then showing that any parallel-prefix graph with depth k and n inputs must have size at least

$$\Omega(n \log^{\overbrace{* \cdot \cdot \cdot *}^{k-2}} n).$$

That this result has no implications for nonmonotone networks can be seen as follows. Consider prefix-or, the parallel-prefix problem for the "or" semigroup (the semigroup with elements 0 and 1 and "or" as its operation). A monotone network computing prefix-or must also contain a parallel-prefix graph, so the lower bound of

$$\Omega(n \log^{\overbrace{* \cdot \cdot \cdot *}^{k-2}} n)$$

also applies to such networks. But Chandra, Fortune, and Lipton [18] have shown that prefix-or can be computed by nonmonotone networks with constant delay and linear cost (the reader may enjoy proving this as an exercise). Thus the concept of a parallel-prefix graph cannot be used to prove lower bounds for nonmonotone networks.

Chandra, Fortune, and Lipton [17] also observed, however, that an adder or a (not necessarily monotone) network computing carries must contain another type of graph called a "weak superconcentrator." Dolev, Dwork, Pippenger, and Wigderson [19] then proved that a weak superconcentrator with n inputs and depth $2k$ must have size at least

$$\Omega(n \log^{\overbrace{* \cdot \cdot \cdot *}^{k-1}} n).$$

Thus neither addition nor carry computation can be performed by networks with both constant delay and linear cost. This result, together with the upper bound, shows that the minimum possible delay for adders of linear size is $O(\log^{\#} n)$, where

$$\log^{\#} n = \min\{k \ge 0 : \log^{\overbrace{* \cdot \cdot \cdot *}^{k}} n \le k\}.$$

The example of prefix-or shows that negation may be used to reduce the complexity of computations that can be performed without it. A particularly dramatic example of this phenomenon is due to Razborov [20], who shows that the monotone "logical permanent" function, which can be computed by networks with polynomial cost, can be computed by monotone networks only with cost $n^{\Omega(\log n)}$.

3. Multiplication in binary

Suppose that we are given the binary representations $x_0, \cdots, x_{n-1}$ and $y_0, \cdots, y_{n-1}$ of two n-bit binary numbers x and y, and that we wish to compute the binary representation $z_0, \cdots, z_{2n-1}$ of their $2n$-bit product $z = xy$. As for addition, since the most significant output depends on all the inputs, a network with bounded fan-in must have cost $\Omega(n)$ and delay $\Omega(\log n)$.

To get our bearings, let us consider the most obvious method for multiplication. First, compute the binary representations of the n "partial products" $x_0 y$, $2x_1 y$, $\cdots$, $2^{n-1}x_{n-1}y$. Then, add these to form the "total product" z. The formation of the partial products is trivially accomplished with cost $O(n^2)$ and delay $O(1)$. The accumulation of the total product can be accomplished by adding the partial products in pairs, adding the results in pairs, and so forth, until a single result is obtained. The resulting "tree" of adders contains $n - 1$ adders, of which at most $\lceil \log_2 n \rceil$ lie on any path from a "leaf" (partial product) to the "root" (total product). As we saw in Section 2, each adder can be constructed with cost $O(n)$ and delay $O(\log n)$, so the resulting multiplier has cost $O(n^2)$ and delay $O((\log n)^2)$.

There is a useful device, called "carry-save addition," that may be brought to bear here. In the present instance, it reduces the delay from $O((\log n)^2)$ to $O(\log n)$, without affecting the cost. It is applicable whenever a large number of additions must be performed, and in particular it is applicable to other multipliers, with smaller costs, that are described later in this section.

The idea behind carry-save addition is the use of a redundant representation for numbers. Instead of using the ordinary binary representation $x_0, \cdots, x_{n-1}$ for x, let us agree to use any combination of $2n$ bits $x'_0, x''_0, \cdots, x'_{n-1}, x''_{n-1}$ such that

$$x = \sum_{0\le m\le n-1} (x'_m + x''_m)2^m.$$

The representation is now no longer unique, but precisely this flexibility makes it easier to perform additions.

Suppose we are given the redundant representations $x'_0, x''_0, \cdots, x'_{n-1}, x''_{n-1}$ and $y'_0, y''_0, \cdots, y'_{n-1}, y''_{n-1}$ for numbers x and y and that we wish to compute a redundant representation $z'_0, z''_0, \cdots, z'_n, z''_n$ for their sum $z = x + y$.

The rules for addition symmetrically combine three bits (denoted x_m, y_m, and c_m in Section 2) to obtain one bit associated with the same position (denoted z_m) and one bit associated with the next higher position (denoted c_{m+1}). Let us combine x'_m, x''_m, and y'_m in this way to obtain a_m and b_{m+1}. Of course, b_m is produced by the next lower position. Let us then combine a_m, b_m, and y''_m in the same way to obtain a'_m and b'_{m+1}. Finally, we set $z'_m = a'_m$ and $z''_m = b'_m$ (which is produced by the next lower position). This method actually produces a bit z''_{n+1} that has no place in our result, but if x and y are at most $2^n - 1$, this extra bit is 0 and need not be computed. Thus carry-save addition can be performed by networks with cost $O(n)$ and delay $O(1)$.

If we now construct a multiplier as before, but employ carry-save adders instead of ordinary adders, we obtain a multiplier with cost $O(n^2)$ but delay only $O(\log n)$. Of course, this multiplier accepts its inputs and produces its outputs in redundant representation, but this difficulty is easily overcome. To convert inputs from ordinary binary to redundant binary, we may set $x'_m = x_m$ and $x''_m = 0$. To convert the output from redundant binary to ordinary binary, we may employ an ordinary adder; this contributes $O(n)$ to the cost and $O(\log n)$ to the delay, yielding an ordinary multiplier with cost $O(n^2)$ and delay $O(\log n)$.

Having seen how to multiply with the minimum possible delay, let us now turn to the problem of reducing the cost. The first step was taken by Karatsuba [21] in 1962; he showed how to construct multipliers with cost $O(n^{\log_2 3})$ ($\log_2 3 = 1.585 \cdots$). One way of doing this (more suitable for generalization than the one used by Karatsuba) is as follows. Suppose that $n = 2n'$ is even (the case of n odd is similar). Regard the input x as the value $p(2^{n'})$ of a linear polynomial $p(\xi) = p_0 + p_1\xi$ at the point $\xi = 2^{n'}$, where the coefficients p_0 and p_1 are n'-bit numbers. Regard y similarly as the value $q(2^{n'})$ of a polynomial $q(\xi) = q_0 + q_1\xi$. If we can compute the product $r(\xi) = p(\xi)q(\xi)$, which is a quadratic polynomial, then z can be obtained as its value $r(2^{n'})$. Since a quadratic polynomial is determined by its values at any three distinct points, one way of computing $r(\xi)$ is to evaluate $p(\xi)$ and $q(\xi)$ at three points, say $\xi \in \{0, 1, 2\}$; multiply these values in pairs (by recursive application of the procedure being described) to obtain $r(0)$, $r(1)$, and $r(2)$; then interpolate to obtain the coefficients r_0, r_1, and r_2 of $r(\xi) = r_0 + r_1\xi + r_2\xi^2$. The values of $p(\xi)$ and $q(\xi)$ at $\xi \in \{0, 1, 2\}$ are at most $7(2^{n'} - 1)$, and thus are $(n' + 3)$-bit numbers. Thus, the multiplication of a pair of n-bit numbers is reduced to the multiplication of three pairs of $(n' + 3)$-bit numbers, together with some evaluation and interpolation operations. The evaluation and interpolation reduce to shifting, addition, and subtraction, and if arithmetic is done modulo $2^{2n} + 1$, then 2^{2n} serves as -1, so subtraction reduces to cyclic shifting and addition. The recursion reduces n by about a factor of 2, while increasing the number of problems by a factor of 3, at a cost of $O(n)$. It follows that the final cost is $O(n^{\log_2 3})$.

The foregoing scheme may be generalized by employing polynomials of higher degree and evaluating and interpolating at more points. This was done by Toom [22], whose scheme when optimized yields a cost bound of the form $O(n2^{\sqrt{2\log_2 n}} \log n)$. Independently, Schönhage [23] proposed a different scheme, based on modular arithmetic but yielding the similar cost bound $O(n2^{\sqrt{2\log_2 n}}(\log n)^{3/2})$. It was Schönhage and Strassen [24] in 1971 who obtained the cost bound $O(n \log n \log \log n)$, which remains the best that is known today. Their idea is to evaluate and interpolate polynomials at many points, but to exploit a particularly advantageous choice of points. If 2^m points are used, and if the points are chosen to be the 2^mth roots of unity, the evaluation and interpolation processes become the computation of a finite Fourier transform and its inverse, for which a particularly efficient algorithm (the "Fast Fourier Transform," due to Cooley and Tukey [25]) is available. As a final touch of elegance, Schönhage and Strassen observe that if arithmetic is done modulo a number of the form $2^{2^{m-1}} + 1$, then 2 serves as a 2^mth root of unity, so Fourier transforms can be done without complex numbers. This modulus is also convenient for the use of redundant representation to reduce delay; combining all these ideas yields a multiplier with delay $O(\log n)$ and cost $O(n \log n \log \log n)$.

It remains an open question whether there are multipliers of linear cost. It is widely believed that there are not, but no nonlinear bounds to the cost of multipliers (or, as was noted in Section 1, nonmonotone networks computing any other "simple" functions) have yet been obtained.

For networks with unbounded fan-in, it is not at all clear how to improve the delay bound of $O(\log n)$ significantly without an enormous increase in cost. In particular, it is not clear whether bounded delay and polynomial cost can be achieved simultaneously. It is natural in this situation to look for a simple problem that captures the essence of multiplication in the same way that carry computation captures the essence of addition. We show that "majority" is such a problem.

Let us say that a problem (with a parameter n) is "easy" if it can be computed by a network with unbounded fan-in, bounded delay (independent of n), and polynomial cost (as a function of n). We perform a cycle of reductions among problems. First, we show that if multiplication is easy, then so is majority (with n arguments). Then we show that if majority is easy, then so is "counting" (that is, determining the binary representation of the number of 1s among its n arguments). Finally, we close the cycle by showing that if counting is easy, then so is multiplication. This establishes the equivalence of multiplication and majority.

First let us show that if multiplication is easy, then so is majority. Suppose that we wish to determine the number of 1s among $n = 2^\nu - 1$ Boolean arguments $x_0, \cdots, x_{n-1}$. This number is the coefficient of ξ^{n-1} in the product of the polynomials $x(\xi) = x_0 + x_1\xi + \cdots + x_{n-1}\xi^{n-1}$ and $y(\xi) = 1 + \xi + \cdots + \xi^{n-1}$. If we evaluate these polynomials at $\xi = 2^\nu$ and multiply the resulting $(n\nu)$-bit numbers together, the coefficient in question can be read off from the $(n\nu)$th through the $((n + 1)\nu - 1)$st positions of the product, and the most significant of these is the majority. The formation of the factors and the interpretation of the product require no gates, so the delay and cost needed to compute the majority of n variables are at most those needed to multiply two numbers of length $O(n \log n)$. Thus if multiplication is easy, so is majority.

Now let us show that if majority is easy, so is counting. Given a network computing the majority of $2n + 1$ arguments, we can by substituting constants for $n + 1$ of these arguments obtain any "threshold" function of the remaining n arguments (the mth threshold function assumes the value 1 if and only if at least m of its arguments assume the value 1). From the mth and $(m + 1)$st threshold functions we can easily compute the mth "indicator" function (which assumes the value 1 if and only if exactly m of its arguments assume the value 1). Furthermore, from the first through nth indicator functions, we can easily compute any "symmetric" function (that is, any function whose value depends only on the number of 1s among its arguments). But all of the counting functions are symmetric functions. Thus if majority is easy, so is counting.

Finally let us show that if counting is easy, so is multiplication. Suppose we are given two n-bit numbers and we wish to compute their product. We have seen that this can be accomplished by adding together n partial products of length $2n$. Let us begin by counting the number of 1s in each of the $2n$ positions; this gives $2n$ counts, each of length $O(\log n)$. Since each position appears in just $O(\log n)$ of the counts, what remains is the problem of adding together $O(\log n)$ "second" partial products of length $2n$. Repeating this procedure reduces the problem to adding together $O(\log \log n)$ "third" partial products of length $2n$. Now partition the positions into blocks each containing $\lceil \log_2 \log_2 n \rceil$ consecutive positions, and alternately assign the colors "red" and "blue" to the blocks. Let p denote the sum of the third partial products when the red positions in these partial products are set to 0s, and let q denote the sum when the blue positions are set to 0s. It is easy to see that each bit of the binary representation of p or q depends on only $O(\log \log n)$ bits of each of the third partial products, and thus on only $O((\log \log n)^2)$ bits altogether. Thus each position of p or q can easily be computed from the third partial products. Since the final product is the sum of p and q, it can easily be computed from them by an adder. Thus if counting is easy, so is multiplication.

Having shown the equivalence of multiplication and majority, we are left with the question as to whether either of them is easy. This question was answered independently by Furst, Saxe, and Sipser [26] and by Ajtai [27], who showed that no network with bounded delay and polynomial cost can compute parity (with n arguments). As we have just seen, such a negative result for a symmetric function such as parity implies one for majority, and thus for multiplication. The ideas in these two papers, if carried to their limits, yield lower bounds of $n^{\Omega(\log n)}$ for the cost of networks computing parity with bounded delay. This is far from the best upper bounds known, $2^{O(n^{1/(k-1)})}$ for parity and $2^{O(n^{1/(k-1)}(\log n)^{1-1/(k-1)})}$ for majority, for networks with delay k.

This gap was partially closed by Boppana [28], who showed that monotone networks computing majority with delay k must have cost $2^{\Omega(n^{1/(k-1)})}$, nearly matching the upper bound above (which does, in fact, apply to monotone networks). An example due to Ajtai and Gurevich [29], however, shows that there are monotone functions computed by networks of bounded delay and polynomial cost for which any monotone network of bounded delay must have cost $n^{\Omega(\log\log n)}$. This shows that Boppana's result has no implications for nonmonotone networks.

For nonmonotone networks, the decisive step was taken by Yao [30], who showed that networks computing parity with delay k must have cost $2^{\Omega(n^{1/6k})}$. An improvement due to Hastad [31] (which has the added merit of a drastically simplified proof) shows they must have cost $2^{\Omega(n^{1/k})}$, almost matching the upper bound. Hastad's result, together with the upper bound, shows that the minimum possible delay for multipliers of polynomial size is $O(\log n/\log\log n)$.

If a problem is easy, there is some minimum delay for which it is computed by networks with polynomial cost. Sipser [32] showed that for every $k \geq 1$ there is a problem computed by networks with delay $k + 1$ and linear cost, but not by networks with delay k and polynomial cost. Klawe, Paul, Pippenger, and Yannakakis [33] showed that these functions, which are monotone, are computed by monotone networks with delay k and cost about the same as that for parity, and they gave a matching lower bound for monotone networks. The methods of Yao [30] and Hastad [31] give an almost matching lower bound for nonmonotone networks.

The reader may have noticed that although multiplication was shown to be equivalent to majority, the final lower bound was obtained through parity. Parity is a computation in a semigroup (indeed, in a group, the group of integers modulo 2). This prompts us to resume the study of computations in semigroups begun in Section 2.

First, let us mention that the results concerning parity described above all apply equally to the problem "congruence modulo p," for any fixed modulus $p \geq 2$. Thus, computations in cyclic groups are not easy. Since every group contains a cyclic subgroup, computations in groups are not easy. In particular, computations in semigroups that contain a group as a subsemigroup are not easy. The result of Chandra, Fortune, and Lipton [17] shows that computations in finite semigroups that do not contain a group as a subsemigroup are easy, so we now know (at least at the level of "easy" versus "not easy") the complexity of computations in finite semigroups for networks with unbounded fan-in.

Next, let us consider the relationship between parity and majority. Razborov [34] has shown that even if gates are available that compute parity for k arguments with cost k and delay 1, majority is still not easy. An improvement due to Smolensky [35] shows that even if gates are available that compute congruence modulo a prime q for k arguments with cost k and delay 1, congruence modulo p is not easy for any prime p distinct from q. (Taking $p = 2$ and $q = 3$ yields an alternative proof of the result of Yao and Hastad for parity.) On the other hand, Barrington [36] has shown that there is a finite group (the alternating group A_5 of even permutations of 5 points) with the property that, if gates are available that compute the product of k elements with cost k and delay 1, then every problem solved by circuits with bounded fan-in and logarithmic depth (in particular, every problem considered in this paper) is easy.

4. Conclusion

The morals of our study of addition are as follows. The essence of addition is the computation of the carries. This computation is an example of a prefix problem for a finite semigroup, the "carry" semigroup. For networks with bounded fan-in, all finite semigroups have the same complexity, with linear cost and logarithmic delay. For networks with unbounded fan-in, the situation is more complicated. For some semigroups, for example the "or" semigroup, linear cost and logarithmic delay are achievable simultaneously. But the carry semigroup is not among these; for it these bounds are not achievable simultaneously.

The morals of our study of multiplication are as follows. The essence of multiplication is computation of the majority. This is not a semigroup computation, but it is connected to the computation of parity, which is. For some semigroups, for example the carry semigroup, polynomial cost and bounded delay can be achieved simultaneously. But parity is not among these; for it these bounds are not achievable simultaneously.

It would be natural to continue our study of arithmetic operations to division, extraction of square roots, and other operations. We have not done so because this leads away from the algebraic methods that have been emphasized in this paper towards analytic methods. We shall only mention that for networks with bounded fan-in, the best cost bounds known for these problems are the same as those for multiplication [currently, $O(n \log n \log\log n)$], but are achieved by networks with delay $O((\log n)^2)$ through the use of Newton's method (see Cook [37]). This was the best delay known for many years, until a breakthrough by Reif [39], giving delay $O(\log n\,(\log\log n)^2)$. This was quickly followed by an improvement due to Beame, Cook, and Hoover [39], giving delay $O(\log n)$, achieved by networks with cost $O(n^4)$. It is a tantalizing open problem to reconcile these methods to achieve cost $O(n \log n \log\log n)$ and delay $O(\log n)$ simultaneously.

References

1. S. Winograd, "On the Time Required to Perform Addition," *J. ACM* **12,** 277–285 (1965).

2. S. Winograd, "On the Time Required to Perform Multiplication," *J. ACM* **14**, 793–802 (1967).
3. D. E. Muller, "Complexity in Electronic Switching Circuits," *IRE Trans. Electr. Comp.* **5**, 15–19 (1956).
4. H. J. Hoover, M. M. Klawe, and N. J. Pippenger, "Bounding Fan-Out in Logical Networks," *J. ACM* **31**, 13–18 (1984).
5. M. S. Paterson and I. Wegener, "Nearly Optimal Hierarchies for Network and Formula Size," *Acta Informat.* **23**, 217–221 (1986).
6. A. E. Andreev, "On the Complexity of Monotone Functions," *Vest. Mosk. Univ. Mat. Mekh.* **1**, No. 4, 83–87 (1985).
7. A. A. Razborov, "Lower Bounds for the Monotone Complexity of Some Boolean Functions," *Dokl. Akad. Nauk* **281**, 798–801 (1985).
8. A. E. Andreev, "On One Method of Obtaining Lower Bounds of Individual Monotone Function Complexity," *Dokl. Akad. Nauk* **282**, 1033–1037 (1985).
9. M. Nadler, "A High-Speed Electronic Arithmetic Unit for Automatic Computing Machines," *Acta Techn.* **16**, 464–478 (1956).
10. A. Weinberger and J. L. Smith, "A One-Microsecond Adder Using One-Megacycle Circuitry," *IRE Trans. Electr. Comp.* **5**, 67–73 (1956).
11. J. Sklansky, "Conditional Sum Addition Logic," *IRE Trans. Electr. Comp.* **9**, 226–231 (1960).
12. Yu. P. Ofman, "The Algorithmic Complexity of Discrete Functions," *Sov. Phys. Dokl.* **7**, 589–591 (1963).
13. V. M. Khrapchenko, "Asymptotic Estimation of Addition Time of a Parallel Adder," *Prob. Kibernet.* **19**, 107–122 (1967).
14. R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," *J. ACM* **27**, 831–838 (1980).
15. F. E. Fich, "New Bounds for Parallel Prefix Circuits," *ACM Symp. Theor. Computing* **15**, 100–109 (1983).
16. M. Snir, "Depth-Size Trade-Offs for Parallel Prefix Computation," *J. Algor.* **7**, 185–201 (1986).
17. A. K. Chandra, S. J. Fortune, and R. J. Lipton, "Unbounded Fan-In Circuits and Associative Functions," *ACM Symp. Theor. Computing* **15**, 52–60 (1983).
18. A. K. Chandra, S. J. Fortune, and R. J. Lipton, "Lower Bounds for Constant Depth Monotone Circuits for Prefix Functions," *Int. Colloq. Automata, Languages & Programming* **10**, 109–117 (1983).
19. D. Dolev, C. Dwork, N. Pippenger, and A. Wigderson, "Superconcentrators, Generalizers and Generalized Connectors with Limited Depth," *ACM Symp. Theor. Computing* **15**, 42–51 (1983).
20. A. A. Razborov, "A Lower Bound to the Monotone Complexity of the Logical Permanent," *Mat. Zametki* **37** (1985).
21. A. Karatsuba and Yu. Ofman, "Multiplication of Multidigit Numbers on Automata," *Sov. Phys. Dokl.* **7**, 595–596 (1963).
22. A. L. Toom, "The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers," *Sov. Math.* **3**, 714–716 (1963).
23. A. Schönhage, "Multiplikation großer Zahlen," *Computing* **1**, 182–196 (1966).
24. A. Schönhage and V. Strassen, "Schnelle Multiplikation großer Zahlen," *Computing* **7**, 281–292 (1971).
25. J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Computation of Complex Fourier Series," *Math. Comp.* **19**, 297–301 (1965).
26. M. Furst, J. B. Saxe, and M. Sipser, "Parity, Circuits and the Polynomial-Time Hierarchy," *IEEE Symp. Found. Comp. Sci.* **22**, 260–270 (1981).
27. M. Ajtai, "Σ_1^1-Formulae on Finite Structures," *Ann. Pure Appl. Logic* **24**, 1–48 (1983).
28. R. Boppana, "Threshold Functions and Bounded Depth Monotone Circuits," *ACM Symp. Theor. Computing* **16**, 475–479 (1984).
29. M. Ajtai and Y. Gurevich, "Monotone Versus Positive," *J. ACM*, to appear.
30. A. C.-C. Yao, "Separating the Polynomial-Time Hierarchy by Oracles," *IEEE Symp. Found. Comp. Sci.* **26**, 1–10 (1985).
31. J. Hastad, "Almost Optimal Lower Bounds for Small Depth Circuits," *ACM Symp. Theor. Computing* **18**, 6–20 (1986).
32. M. Sipser, "Borel Sets and Circuit Complexity," *ACM Symp. Theor. Computing* **15**, 61–69 (1983).
33. M. Klawe, W. Paul, N. Pippenger, and M. Yannakakis, "On Monotone Formulae with Restricted Depth," *ACM Symp. Theor. Computing* **16**, 480–487 (1984).
34. A. A. Razborov, "Lower Bounds on the Size of Bounded-Depth Networks Over the Basis $\{\wedge, \oplus\}$," *Mat. Zametki*, to appear.
35. R. Smolensky, "Algebraic Methods in the Theory of Lower Bounds for Boolean Circuit Complexity," Department of Mathematics, University of California at Berkeley, 1986.
36. D. A. Barrington, "Bounded-Width Polynomial-Size Branching Programs Recognize Exactly Those Languages in NC^1," *ACM Symp. Theor. Computing* **18**, 1–5 (1986).
37. S. A. Cook, "On the Minimum Computation Time of Functions," Thesis, Harvard University, Cambridge, MA, 1966.
38. J. H. Reif, "Logarithmic Depth Circuits for Algebraic Functions," *IEEE Symp. Found. Comp. Sci.* **24**, 138–145 (1983).
39. P. W. Beame, S. A. Cook, and H. J. Hoover, "Log Depth Circuits for Division and Related Problems," *IEEE Symp. Found. Comp. Sci.* **25**, 1–6 (1984).

Received December 18, 1986; accepted for publication January 19, 1987

Nicholas Pippenger *IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120.* Dr. Pippenger received a B.S. in natural science from Shimer College, Mt. Carroll, Illinois, in 1965 and the B.S., M.S., and Ph.D. degrees in electrical engineering from the Massachusetts Institute of Technology in 1967, 1969, and 1973. From 1969 to 1972, he was a staff member of the Draper Laboratory (formerly the MIT Instrumentation Laboratory) in Cambridge. He joined the Mathematical Sciences Department of the IBM Thomas J. Watson Research Center in 1973 and served as the department's assistant director from 1975 to 1976. In 1978 he served as a visiting associate professor in the Computer Science Department at the University of Toronto, Canada. He returned as manager of the theoretical computer sciences group and served in that capacity from 1979 to 1980, at which time he transferred to the Computer Science Department at San Jose.