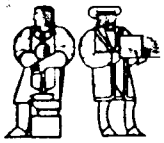
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-151

REVERSIBLE COMPUTING

Tommaso Toffoli

February 1980

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

REVERSIBLE COMPUTING*

Tommaso Toffoli

MIT Laboratory for Computer Science
545 Technology Sq., Cambridge, MA 02139

Abstract. The theory of reversible computing is based on invertible primitives and composition rules that preserve invertibility. With these constraints, one can still satisfactorily deal with both functional and structural aspects of computing processes; at the same time, one attains a closer correspondence between the behavior of abstract computing systems and the microscopic physical laws (which are presumed to be strictly reversible) that underly any concrete implementation of such systems.

Here, we integrate into a comprehensive picture a variety of concepts and results. According to a physical interpretation, the central result of this paper is that *it is ideally possible to build sequential circuits with zero internal power dissipation*. Even when these circuits are interfaced with conventional ones, power dissipation at the interface would be at most proportional to the number of input/output lines, rather than to the number of logic gates as in conventional computers.

Keywords. Reversible computing, computation universality, automata, computing networks, physical computing.

1. Introduction

Mathematical models of computation are abstract constructions, by their nature unfettered by physical laws. However, if these models are to give indications that are relevant to concrete computing, they must somehow capture, albeit in a selective and stylized way, certain general physical restrictions to which all concrete computing processes are subjected. For instance, the Turing machine, which embodies in a heuristic form the axioms of computability theory, avowedly accounts in its design[24] for the fact that the speed of propagation of information is bounded, and that the amount of information which can be encoded in the state of a finite system is bounded. However, other physical

*This research was supported by Grant N00014-75-C-0661, Office of Naval Research, funded by DARPA.

principles of comparable importance, such as the reversibility at a microscopic level of the dynamical laws (which imposes severe constraints on the operation of concrete computing primitives[12,23]) and the fact that the topology of spacetime is locally Euclidean (which severely limits the range of interconnection patterns for concrete computing structures[19]), are not yet adequately represented in the theory of computing. Conceivably, a better match between the abstract constructs of the theory and its applications would be attained if a suitable counterpart of these principles were incorporated in the theory.

Here, we shall be concerned with the issue of *reversibility*. Intuitively, a dynamical system is reversible if from any point of its state set one can uniquely trace a trajectory backward as well as forward in time. For a time-discrete system such as an automaton, this is equivalent to saying that its transition function is invertible, that is, bijective. The concept of reversibility has its origins in physics, and, in particular, in the study of continuous systems, such as those characterized by a differential equation, rather than discrete systems, which are characterized by a transition function. In the continuous case, a more technical definition is necessary; namely, a dynamical system is *reversible* if its dynamical semigroup can be expanded to a group[14]. The connection between these two definitions is immediate, since in the discrete case the transition function coincides with the generator of the system's semigroup.

It should be noted that reversibility does not imply invariance under time reversal; the latter is a more specialized notion which is definable in a nontrivial way only for dynamical systems having additional structure.

In the past, some misgivings were expressed concerning the computing capabilities of reversible automata. Such misgivings were cleared by the work of Bennett (reversible Turing machines[4]), Priese (reversible computers embedded in Thue systems[17]), Fredkin (conservative logic[7]), and Toffoli (reversible cellular automata[20]). Today, the concept of reversible computing appears to be not only productive from a theoretical viewpoint but also promising in terms of technological applications[8].

In fact, one of the strongest motivations for the study of reversible computing comes from the desire to reduce heat dissipation in computing machinery, and thus achieve higher density and speed. Briefly, while the laws of physics are presumed to be strictly reversible, abstract computing is usually thought of as an irreversible process, since it may involve the evaluation of many-to-one functions. Thus, as one proceeds down from an abstract computing task to a formal realization by means of a digital network and finally to an implementation in a physical system, at some level of this modeling hierarchy there must take place the transition from the irreversibility of the given computing process to

the reversibility of the physical laws. In the customary approach, this transition occurs at a very low level and is hidden—so to speak—in the "physics" of the individual digital gate;* as a consequence of this approach, the details of the work-to-heat conversion process are put beyond the reach of the conceptual model of computation that is used.

On the other hand, it is possible to formulate a more general conceptual model of computation such that the gap between the irreversibility of the desired behavior and the reversibility of a given underlying mechanism is bridged *in an explicit way* within the model itself. This we shall do in the present paper.

An important advantage of our approach is that any operations (such as the clearing of a register) that in conventional logic lead to the destruction of macroscopic information, and thus entail energy dissipation, here can be planned at the whole-circuit level rather than at the gate level, and most of the time can be replaced by an information-lossless variant. As a consequence, *it appears possible to design circuits whose internal power dissipation, under ideal physical circumstances, is zero.* The power dissipation that would arise at the interface between such circuits and the outside world would be at most proportional to the number of input/output lines, rather than to the number of logic gates.

2. Terminology and notation

A function $\phi: X \to Y$ is *finite* if X and Y are finite sets. A *finite automaton* is a dynamical system characterized by a transition function of the form $\tau: X \times Q \to Q \times Y$, where τ is finite.

Without loss of generality, one may assume that such sets as X, Y, and Q above be explicitly given as indexed Cartesian products of sets. We shall occasionally call *lines* the individual variables associated with the individual factors of such products. By convention, the Cartesian product of zero factors is identified with the *dummy* set $\{\lambda\}$ consisting of the *empty* word λ. In what follows, we shall assume *once and for all* that all factors of the aforementioned Cartesian products be identical copies of the *Boolean* set $B = \{0, 1\}$. This assumption entails little loss of generality, and—at any rate—the theory of reversible computing could be developed along essentially the same lines if such assumption were dropped.

*Typically, the computation is logically organized around computing primitives that are not invertible, such as the NAND gate; in turn, these are realized by physical devices which, while by their nature obeying reversible microscopic laws, are made *macroscopically* irreversible by allowing them to convert some work to heat.

The concept of "function composition" is a fundamental one in the theory of computing. According to the ordinary rules for function composition, an output variable of one function may be substituted for *any number* of input variables of other functions, i.e., arbitrary "fan-out" of lines is allowed. However, the process of generating multiple copies of a given signal must be treated with particular care when reversibility is an issue (moreover, from a physical viewpoint this process is far from trivial). For this reason, in all that follows we shall restrict the meaning of the term "function composition" to *one-to-one* composition, where any substitution of output variables for input variables is one-to-one (in other words, no fan-out of lines is allowed). Any fan-out node in a given function-composition scheme will have to be treated as an explicit occurrence of a fan-out function of the form $(x) \mapsto (x, \ldots, x)$. Intuitively, the responsibility for providing fan-out is shifted from the composition rules to the computing primitives.

We shall be dealing with various classes of abstract computers (such as combinational networks, finite automata, Turing machines, and cellular automata) which constitute the main paradigms of the theory of computing. An abstract computer is, in essence, a function-composition scheme,* and a computation is a particular solution (which may be required to satisfy certain boundary conditions or other constraints) of such a scheme. While finite composition schemes are adequate for dealing with the most elementary aspects of computing, many computing processes of interest require an unbounded amount of resources and are more conveniently represented as taking place in infinite schemes.

It is customary to express a function-composition scheme in graphical form as a *causality network* (or *functional-dependence* network). This is basically an acyclic directed graph whose nodes are labeled by associating with each of them a particular finite function and whose arcs are colored by associating with each of them a particular variable. (Here, we can safely ignore certain slight technical differences between a causality network and a directed graph.)

By construction, causality networks are "loop-free," i.e., they contain no cyclic paths. A *combinational* network is a causality network that contains no infinite paths. Note that a finite causality network is always a combinational one.

With certain additional conventions, causality networks having a particular iterative structure can be represented more compactly as *sequential* networks (cf. Section 7).

We shall assume familiarity with the concept of "realization" of finite func-

*In what follows, we shall restrict our attention to function-composition schemes based on finite primitives.

tions and automata by means of, respectively, combinational and sequential networks. In what follows, a "realization" will always mean a *componentwise* one; that is, to each input (or output) line of a finite function there will correspond an input (or output) line in the combinational network that realizes it, and similarly for the realization of automata by sequential networks.

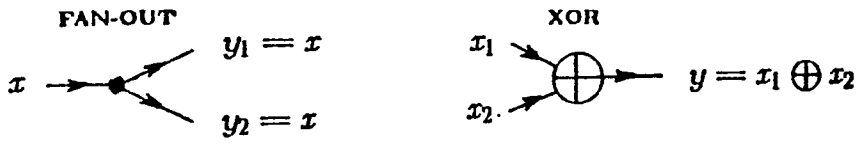
A causality network is *reversible* if it is obtained by composition of invertible primitives. Note that a reversible combinational network always defines an invertible function. Thus, in the case of combinational networks the structural aspect of "reversibility" and the functional aspect of "invertibility" coincide. A sequential network is *reversible* if its *combinational part* (i.e., the combinational network obtained by deleting the delay elements and thus breaking the corresponding arcs) is reversible.

3. Introductory concepts

As explained in Section 1, our overall goal is to develop an explicit realization of computing processes within the context of reversible systems. For the moment, the processes we shall deal with will be those described by finite functions, and the systems used for their realization will be reversible *combinational* networks. Later on, we shall consider sequential processes, characterized by the presence of internal states in addition to input and output states, and we shall study their realization by means of reversible *sequential* networks.

As an introduction, let us consider two simple functions, namely, FAN-OUT (3.1a) and XOR (3.1b):

$$\text{(a)}\quad \begin{array}{c|cc} x & y_1 & y_2 \\ \hline 0 & 0 & 0 \\ 1 & 1 & 1 \end{array} \qquad \text{(b)}\quad \begin{array}{cc|c} x_1 & x_2 & y \\ \hline 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{array} \tag{3.1}$$



Neither of these functions is invertible. (Indeed, FAN-OUT is not *surjective*, since, for instance, the output $(0, 1)$ cannot be obtained for any input value; and XOR

is not *injective*, since, for instance, the output **0** can be obtained from two distinct input values, $(0,0)$ and $(1,1)$). Yet, both functions admit of an invertible realization.

To see this, consider the *invertible* function XOR/FAN-OUT defined by the table

$$\begin{array}{cc} 00 & 00 \\ 01 & 11 \\ 10 & 10 \\ 11 & 01 \end{array} \qquad (3.2)$$

which we have copied over with different headings in (3.3*a*), (3.3*b*), and (3.6*b*). Then, FAN-OUT can be realized by means of this function[6] as in (3.3*a*) (where we have outlined the relevant table entries), by assigning a value of **0** to the auxiliary input component c; and XOR can be realized by means of the same function as in (3.3*b*), by simply disregarding the auxiliary output component g. In more technical terms, (3.1*a*) is obtained from (3.3*a*) by componentwise *restriction*, and (3.1*b*) from (3.3*b*) by *projection*.

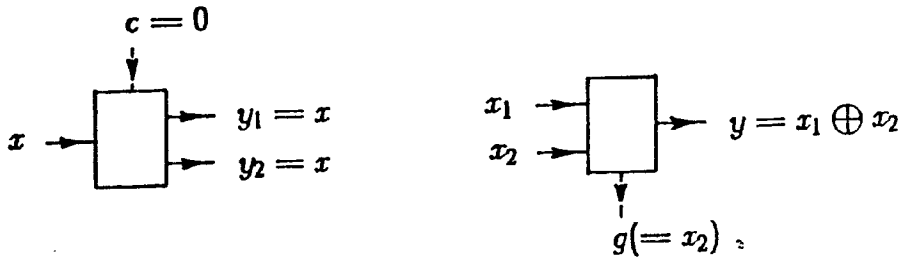
(a)

c	x	y_1	y_2
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	1

(b)

x_1	x_2	y	g
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	1

(3.3)

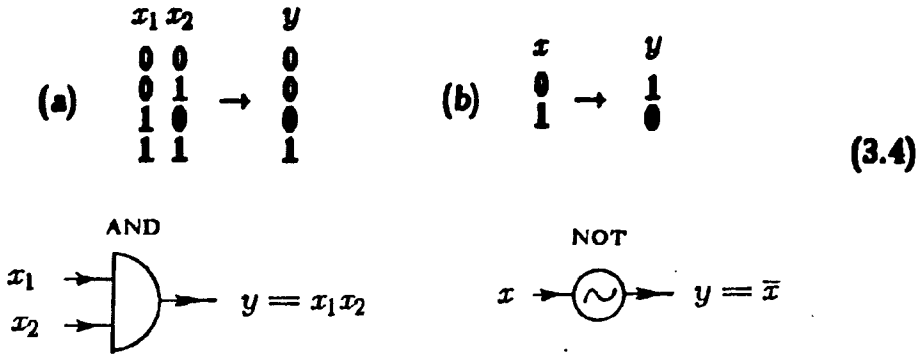


In what follows, we shall collectively call *the source* the auxiliary input components that have been used in a realization, such as component c in (3.3*a*), and *the sink* the auxiliary output components such as g in (3.3*b*). The remaining input components will be collectively called *the argument*, and the remaining output components, *the result*.

In general, both source and sink lines will have to be introduced in order

[6]Ordinarily, one speaks of a realization "by a network." Note, though, that a finite function by itself constitutes a trivial case of combinational network.

to construct an invertible realization of a given function.

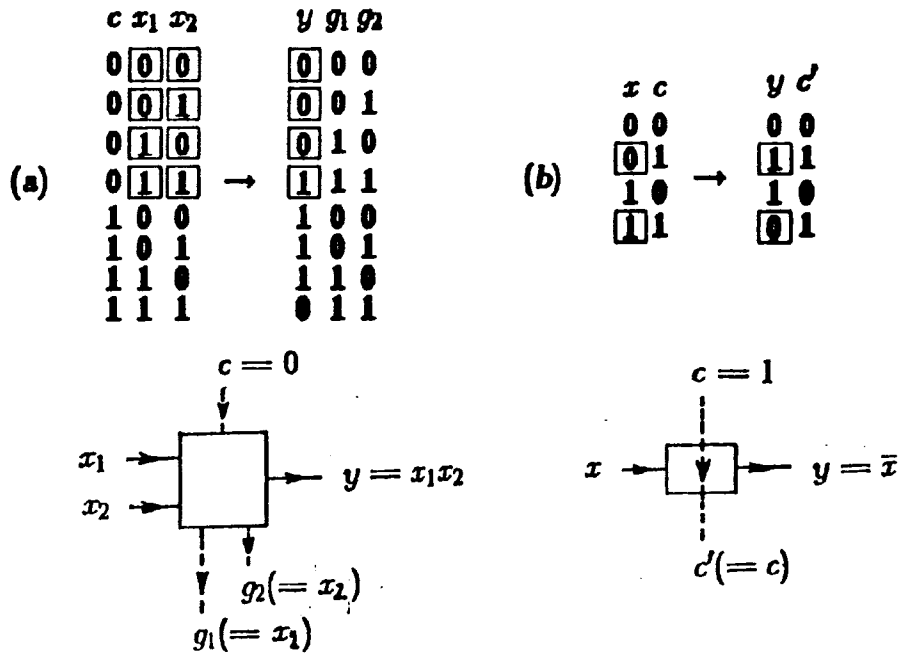


(3.4)

For example, from the *invertible* function AND/NAND defined by the table

$$\begin{array}{ccc} 000 & \to & 000 \\ 001 & \to & 001 \\ 010 & \to & 010 \\ 011 & \to & 011 \\ 100 & \to & 100, \\ 101 & \to & 101 \\ 110 & \to & 110 \\ 111 & \to & 011 \end{array} \tag{3.5}$$

the AND function (3.4a) can be realized as in (3.6a) with one source line and two sink lines.



(3.6)

Observe that in order to obtain the desired result the source lines must be fed with specified *constant* values, i.e., with values that do not depend on the argument. As for the sink lines, some may yield values that do depend on the argument—as in (3.6a)—and thus cannot be used as input constants for a new computation; these will be called *garbage* lines. On the other hand, some sink lines may return constant values; indeed, this happens whenever the functional relationship between argument and result is itself an invertible one. To give a trivial example, suppose that the NOT function (3.4*b*), which is invertible, were not available as a primitive. In this case one could still realize it starting from another invertible function, e.g., from the XOR/FAN-OUT function as in (3.6*b*); note that here the sink, c', returns in any case the value present at the source, c. In general, if there exists between a set of source lines and a set of sink lines an invertible functional relationship that is independent of the value of all other input lines, then this pair of sets will be called (for reasons that will be made clear in Section 5) a *temporary-storage channel*.

Using the terminology just established, we shall say that the above realization of the FAN-OUT function by means of an invertible combinational function is a realization *with constants*, that of the XOR function, *with garbage*, that of the AND function, *with constants and garbage*, and that of the NOT function, with *temporary storage* (for the sake of nomenclature, the source lines that are part of a temporary-storage channel will not be counted as lines of constants). In referring to a realization, features that are not explicitly mentioned will be assumed not to have been used; thus, a realization "with temporary storage" is one *without* constants or garbage. A realization that does not require any source or sink lines will be called an *isomorphic* realization.

4. The fundamental theorem

In the light of the particular examples discussed in the previous section, this section establishes a general method for realizing an *arbitrary* finite function ϕ by means of an *invertible* finite function f.

Here, we are concerned with realizations in the sense defined in Section 2. In more general mathematical parlance, a *realization* of a function ϕ consists of a new function f together with two mappings μ and ν (respectively, the *encoder* and the *decoder*) such that $\phi = \nu f \mu$. In this context, our plan is to obtain a realization $\nu f \mu$ of ϕ such that f is invertible and the mappings μ and ν are essentially independent of ϕ and contain as little "computing power" as possible. More precisely, though the form of μ and ν must obviously reflect the number of input and output components of ϕ, and thus the *format* of ϕ's truth table,

we want them to be otherwise independent of the particular *contents* of such truth table as ϕ is made to range over the set of all combinatorial functions.

In general, given any finite function one obtains a new one by assigning specified values to certain distinguished input lines (*source*) and disregarding certain distinguished output lines (*sink*). According to the following theorem, any finite function can be realized in this way starting from a suitable *invertible* one.

THEOREM 4.1 *For every finite function* $\phi: \mathbf{B}^m \to \mathbf{B}^n$ *there exists an invertible finite function* $f: \mathbf{B}^r \times \mathbf{B}^m \to \mathbf{B}^n \times \mathbf{B}^{r+m-n}$, *with* $r \leq n$, *such that*

$$f(\overbrace{0,\ldots,0}^{r}, x_1, \ldots, x_m) = \phi_i(x_1, \ldots, x_m), \qquad (i = 1, \ldots, n). \tag{4.1}$$

Proof. Let ϕ be defined by a binary table of the following form

$$2^m\{\overbrace{X}^{m} \to \overbrace{Y}^{n},$$

where X denotes a listing of all 2^m m-tuples over B and Y denotes a listing of the corresponding values of ϕ, which are n-tuples over B. We shall define a function $f: \mathbf{B}^{n+m} \to \mathbf{B}^{n+m}$ by means of the following table

2^n blocks	n	m		n	m
	0	X		Y	X
	1	X		$Y+1$	X
	...	...	→	...	...
	2^n-1	X		$Y+2^n-1$	X

,

where each block of the form k $(0 \leq k < 2^n)$ consists of 2^m identical n-tuples each representing the integer k written in base 2, while each block of the form $Y + k$ $(0 \leq k < 2^n)$ consists of the 2^m entries of Y each treated as a base-2 integer and incremented by $k \bmod 2^n$. (So the sequence of "$Y + k$" blocks differs from the "k" sequence only by a circular permutation.) By construction, each side of this table contains each element of $\mathbf{B}^{n+m}$ exactly once. Thus, f is invertible. Moreover, equation (4.1), with $r = n$, holds by construction. ∎

The meaning of Theorem 4.1 is illustrated in Figure 4.1 below. The operations of restriction and projection mentioned in Section 3 are respectively performed by an input encoder μ and an output decoder ν. While letting through the argument $(x_1, \ldots, x_m)$ unchanged, the encoder supplies the r source lines with constant values, i.e., with values that do not depend on the argument itself. On the other hand, while letting through the result $(y_1, \ldots, y_n)$ unchanged, the decoder absorbs whatever values come out of the the $m + r - n$ sink lines.

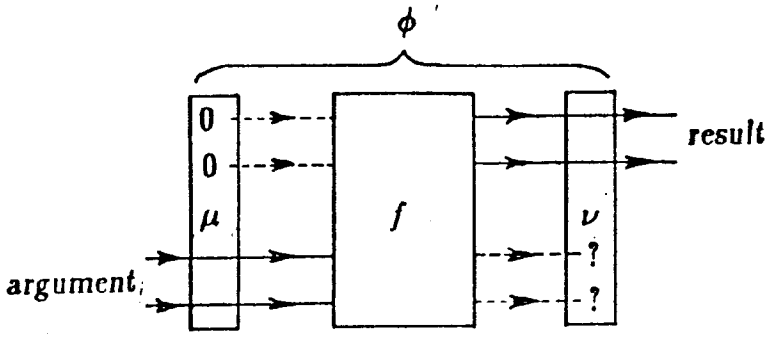


FIG. 4.1 *Any finite function ϕ can be written as the product of a trivial encoder μ, an invertible finite function f, and a trivial decoder ν.*

Intuitively, such a realization of ϕ by means of an invertible function f is "fair," in the sense that neither μ nor ν contribute to the "computing power" of f.

Since μ does not interact with the input signals, it will be more convenient to visualize it as a separate *source* of constant input values, as in Figure 4.2*b* rather than as an *input encoder*, as in Figure 4.1; likewise, ν is more conveniently visualized as a separate *sink* of output values rather than as an *output decoder*. To sum up, whatever can be computed by an arbitrary finite function according to the schema of Figure 4.2*a* can also be computed by an *invertible* finite function according to the schema of Figure 4.2*b*.

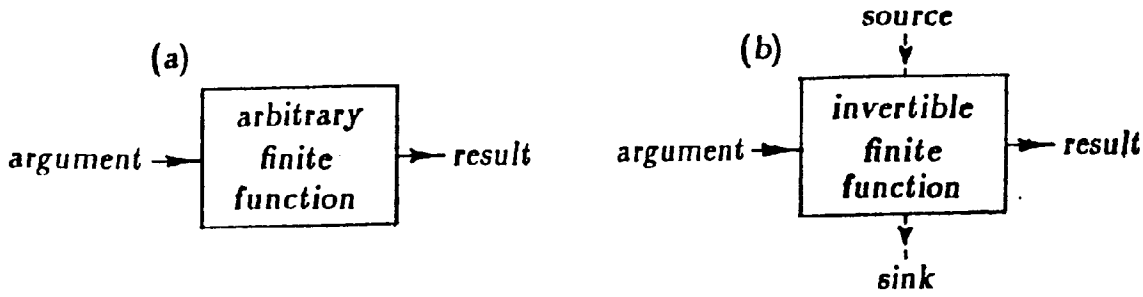


FIG. 4.2 *Any finite function (a) can be realized as an invertible finite function (b) having a number of auxiliary input lines which are fed with constants and a number of auxiliary output lines whose values are disregarded.*

Remark. The construction in the proof of Theorem 4.1 does not necessarily lead to a *minimal* realization, as often the number of source lines can be made strictly less than n and, correspondingly, the number of sink lines strictly less than m. Intuitively, in many cases the given function ϕ is such that much of the information contained in the argument is retained in the result, so that in order to guaranteee invertibility f need only preserve in the garbage signals a *fraction* of the total information.

5. Invertible primitives and reversible networks

In the previous section, each given ϕ was realized by a reversible combinational network consisting of a single occurrence of an *ad hoc* primitive f. In this section, we shall study the realization of arbitrary finite functions by means of reversible combinational networks constructed from given primitives; in particular, from a certain finite set U of very simple primitives.

If the given function ϕ is defined by means of an *arbitrary* combinational network (in what follows, we shall assume for simplicity that this network be based on the NAND element), a reversible realization of ϕ based on the set U can be obtained in a very simple way by subjecting the given network to straightforward translation rules. However, the reversible realization that is obtained in this way in general requires many more sink lines (and, consequently, many more source lines) than the realization of Section 4. On the other hand, starting from the same set U of primitives but using more sophisticated synthesis techniques it is possible to obtain a reversible realization of ϕ that does not require any more garbage lines than when realizing ϕ by means of an *ad hoc* primitive f. In fact, f itself—or, for that matter, any invertible finite function—can be synthesized

from U *without garbage*, though possibly with temporary storage.

In this synthesis, the peculiar constraints dictated by the reversibility context force one to pay attention to a number of issues, such as fan-out, temporary storage, and the handling of constants and garbage, which do not arise—or, at any rate, are not critical—when these constraints are not present. As a counterpart, the structure of the networks that are thus obtained provides more realistic indications of what an efficient physical implementation of the corresponding computing processes would have to be like.

It is well known that, under the ordinary rules of function composition, the two-input NAND element constitutes a universal primitive for the set of all combinational functions.

In the theory of reversible computing, a similar role is played by the AND/NAND element, defined by (3.5) and graphically represented as in Figure 5.1c. Referring to (3.6a), observe that $y = x_1 x_2$ (AND function) when $c = 0$, and $y = \overline{x_1 x_2}$ (NAND function) when $c = 1$. Thus, as long as one supplies a value of 1 to input c and disregards outputs g_1 and g_2, the AND/NAND element can be substituted for any occurence of a NAND gate in an ordinary combinational network.

In spite of having ruled out fan-out as an intrinsic feature provided by the composition rules, one can still achieve it as a *function* realized by means of an invertible primitive, such as the XOR/FAN-OUT element defined by (3.2) and graphically represented as in Figure 5.1*b*. In (3.3a), observe that $y_1 = y_2 = x$ when $c = 0$ (FAN-OUT function); and in (3.3*b*), that $y = x_1 \oplus x_2$ (XOR function).

Finally, recall that finite composition always yields invertible functions when applied to invertible functions (cf. Section 2).

Therefore, using the set of invertible primitives consisting of the AND/NAND element and the XOR/FAN-OUT element, any combinational network can be immediately translated into a reversible one which, when provided with appropriate input constants, will reproduce the behavior of the original network. Indeed, even the set U consisting of the single element AND/NAND is sufficient for this purpose, since XOR/FAN-OUT can be obtained from AND/NAND, with one line of temporarty storage, by taking advantage of the mapping $(1, p, q) \mapsto (1, p, p \oplus q)$.)

The element-by-element substitution procedure outlined above for constructing a reversible-network realization of a given combinational function is *wasteful*, in the sense that the number of source and sink lines that are introduced by this construction is roughly proportional to the number of computing elements that make up the network, and therefore in general much larger than the minimum required to compensate for the noninvertibility of the given function (cf. Section 4).

From the viewpoint of a physical implementation, where signals are encoded in some form of energy, each constant input entails the supply of energy of predictable form, or *work*, and each garbage output entails the removal of energy of unpredictable form, or *heat*. In this context, a realization with fewer source and sink lines might point the way to a physical implementation that dissipates less energy.

Our plan to achieve a less wasteful realization will be based on the following concept. While it is true that each garbage signal is "random," in the sense that it is not predictable without knowing the value of the argument, yet it will be correlated with other signals in the network. Taking advantage of this, one can augment the network in such a way as to make correlated signals interfere with one another and produce a number of *constant* signals instead of garbage. These constants can be used as source signals in other parts of the network. In this way, the overall number of both source and sink lines can be reduced. This process is analogous to the destructive interference of, say, sound waves. It is well-known that by superposing two correlated random signals one may obtain an overall signal which is less "noisy" that either component.

In the remainder of this section we shall show how, in the abstract context of reversible computing, destructive interference of correlated signals can be achieved in a systematic way. For a similar process of destructive interference to take place in concrete computers (thus leading to greatly reduced power dissipation), one would have to match the abstract invertible primitives with digital gates taht are macroscopically—as well as microscopically—reversible. The arguments in [7,8,23] strongly suggest that such gates are indeed physically realizable.

Returning to our mathematical exposition, we shall first show that any invertible finite function can be realized *isomorphically* from certain *generalized* AND/NAND primitives. Then, we shall show that any of these primitives can be realized from the AND/NAND element possibly with temporary storage but with no garbage.

For convenience, we shall say that an invertible finite function is *of order n* if it has n input lines and n output lines.

DEFINITION 5.1 Consider the set $B = \{0,1\}$ with the usual structure of Boolean ring, with "$\oplus$" (*exclusive*-OR) denoting the addition operator, "$\ominus$" the additive-inverse operator (which in this case coincides with the identity operator), and juxtaposition (AND) the multiplication operator. For any $n > 0$, the *generalized* AND/NAND *function of order n*, denoted by $\phi^{(n)}: B^n \to B^n$, is

defined by

$$\theta^{(n)}: \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} \mapsto \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ \ominus x_n \oplus x_1 x_2 \cdots x_{n-1} \end{pmatrix}. \tag{5.1}$$

Remark. (a) The $\ominus$ sign in (5.1), which is redundant (since $\ominus x_n = x_n$), has been introduced for ease of comparison with the arguments of [23]. (*b*) For any $n > 0$, $\theta^{(n)}$ is invertible and coincides with its inverse. (c) For $i = 1, 2, \ldots, n-1$, the i-th component of $\theta^{(n)}$, i.e., $\theta_i^{(n)}$, coincides with the *projection* operator for the corresponding argument, i.e., $\theta_i^{(n)}\langle x_1, \ldots, x_n\rangle = x_i$. (*d*) The last component of $\theta^{(n)}$, i.e., $\theta_n^{(n)}$, coincides with the NOT function for $n = 1$ (note that, by convention, $x_1 \cdots x_i = 1$ when $i = 0$), and with the exclusive-OR of its two arguments for $n = 2$. (e) For all other values of n, $\theta_n^{(n)}$ is still linear in the n-th argument, but is *nonlinear* in the first $n - 1$ arguments.

We have already encountered $\theta^{(1)}$ under the name of the NOT element, $\theta^{(2)}$ under the name of the XOR/FAN-OUT element, and $\theta^{(3)}$ under the name of the AND/NAND element. The generalized AND/NAND functions are graphically represented as in Figure 5.1*d*.

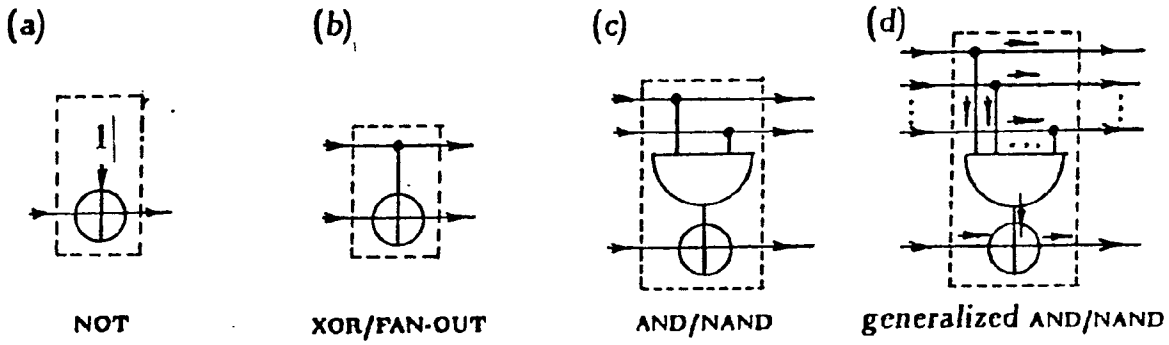


FIG. 5.1 *Graphic representation of the generalized* AND/NAND *functions.* WARNING: This representation is offered only as a mnemonic aid in recalling a function's truth table, and is not meant to imply any "internal structure" for the function, or suggest any particular implementation mechanism. *(a) $\theta^{(1)}$, which coincides with the* NOT *element; (b) $\theta^{(2)}$, which coincides with the* XOR/FAN-OUT *element; (c) $\theta^{(3)}$, which coincides with the* AND/NAND *element; and, in general, (d) $\theta^{(n)}$, the generalized* AND/NAND *function of order n. The bilateral symmetry of these symbols recalls the fact that each of the corresponding functions coincides with its inverse.*

An invertible function acts as a permutation on the elements of its domain, and it is well known that any permutation can be written as a product of *elementary* permutations, i.e., of permutations that exchange exactly two elements. In our attempt to synthesize arbitrary invertible functions of order n we shall consider, as building blocks, elementary permutations on $\mathbf{B}^n$ and even more basic permutations on $\mathbf{B}^n$ called *atomic* permutations.

DEFINITION 5.2 In the truth table for the AND/NAND element (3.5), observe that the only difference between the left- and the right-hand side of the table is that exactly two rows (namely, $\langle 0,1,1\rangle$ and $\langle 1,1,1\rangle$) which have a *Hamming distance* of 1 (i.e., differ in exactly one position) have been exchanged. An *atomic permuter* is any function having this property.

Any atomic permuter of order n can be constructed from the generalized AND/NAND element of order n by appending NOT elements to some of the input lines and to each of the corresponding output lines, as in Figure 5.2a. Graphically, this permuter will be represented as in Figure 5.2b, where some of the inputs to the AND-gate symbol are negated (as denoted by a little circle).

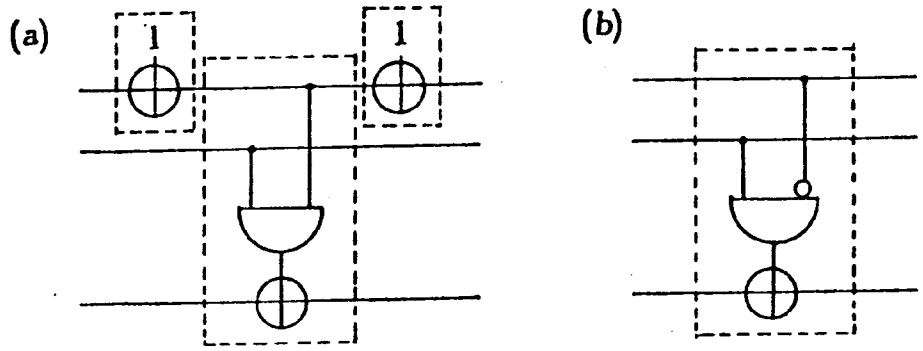


FIG. 5.2 *(a) Construction and (b) graphic representation of a particular atomic permuter.*

THEOREM 5.1 *Any invertible finite function of order n can be obtained by composition of atomic permuters of order n, and therefore by composition of generalized* AND/NAND *functions of order $\leq n$.*

Proof. It will be sufficient to show that one can obtain any elementary permutation on $\mathbf{B}^n$.

In a table of all elements of $\mathbf{B}^n$, a sequence of table entries $a_1, a_2, \ldots, a_i$ (these are all n-tuples) are said to form a *Gray-code path* if any two entries that are adjacent in this sequence are related by an atomic permutation. Consider the pair of entries x and y that we wish to exchange, and consider a Gray-code path

from x to y (such a path exists for any pair x, y). It is easy to verify that by means of sequence of atomic permutations item x can be moved to the end of the path, while the remainder of the path is shifted one position to the left but is otherwise unchanged. In a similar way, y can be brought to the beginning of the path. Thus x and y can be exchanged by means of a sequence of atomic permutations without affecting the rest of the table.∎

Remark. Note that the realization referred to by Theorem 5.1 is an *isomorphic* one (unlike that of Section 4, which makes use of source and sink lines).

THEOREM 5.2 *There exist invertible finite functions of order n which cannot be obtained by composition of generalized* AND/NAND *functions of order strictly less than n.*

Proof. In the context of the proof of Theorem 5.1, when $\theta^{(i)}$ is applied to any i components of B^n this set is divided into 2^{n-i} disjoint collections of 2^i n-tuples, and each collection is permuted by $\theta^{(i)}$ in an identical fashion. Thus, only even permutations can be obtained when $i < n$. Since the product of even permutations is even, only even permutations can be obtained by one-to-one composition of any number of AND/NAND functions of order less than n.∎

Remark. According to this theorem, the AND/NAND primitive is not sufficient for the *isomorphic* reversible realization of arbitrary invertible finite functions of larger order. This result can be generalized to any *finite* set of invertible primitives, as implicit in the proof argument. Thus, one must turn to a less restrictive realization schema involving source and sink lines.

THEOREM 5.3 *Any invertible finite function can be realized, possibly with temporary storage,* [but with no garbage!] *by means of a reversible combinational network using as primitives the generalized* AND/NAND *elements of order* ≤ 3.

Proof. In view of Theorem 5.1, it will be sufficient to realize (possibly with temporary storage), for each n, all atomic permuters of order n. Since these can be realized isomorphically from $\theta^{(1)}$ and $\theta^{(n)}$ (cf. Figure 5.2), it will be sufficient to realize $\theta^{(n)}$ itself. We shall proceed by recursion; namely, given $\theta^{(n-1)}$, $\theta^{(n)}$ can be realized with one line of temporary storage as follows.

Construct the network of Figure 5.3, which contains two occurrences of $\theta^{(n-1)}$ and one occurrence of $\theta^{(3)}$. Observe that $c' \equiv c$, since every generalized AND/NAND element coincides with its inverse (and thus the second occurrence of $\theta^{(n-1)}$ cancels the effect of the first). Therefore, the pair $(\{c\}, \{c'\})$ constitutes a temporary-storage channel. When $c = 0$, the remaining variables behave as the corresponding ones of $\theta^{(n)}$.∎

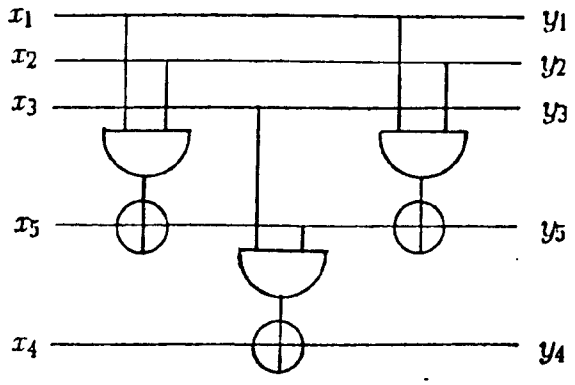


FIG. 5.3 *Realization with temporary storage of $\theta^{(n)}$ from $\theta^{(n-1)}$ (and $\theta^{(3)}$). In this network, when $c = 0$, also $c' = 0$, and the remaining components behave as the corresponding ones of $\theta^{(n)}$.*

In the above construction, it is clear that one line of temporary storage is added every time that one realizes the AND/NAND function of the next higher order. Thence the following theorem.

THEOREM 5.4. *In Theorem 5.3, let n be the order of the given invertible function, and m the number of source (as well as sink) lines required required for temporary-storage channels in the realization. Then m need not exceed $n - 3$. If only $\theta^{(3)}$ is given as a primitive, then m need not exceed $3n - 3$.*

Remark. The second part of this theorem reflects the fact that additional temporary-storage lines are needed to realize NOT from AND/NAND (cf. (3.6)).

The proof of Theorem 5.3 establishes a general mechanisms for bringing about destructive interference of garbage. With reference to Figure 5.3, which can serve as an outline for the general case, observe that the left portion of the network is accompanied by its "mirror image" on the right. The left portion computes an intermediate result (on the line running from c to c') that is needed as an input to the lower portion and is returned by it unchanged. Having performed its function, this intermediate result is then "undone" by the right portion, so that no garbage is left.

The reader may refer to [3,7] for more specific examples of destructive interference of garbage.

Taken together, Theorems 5.3 and 5.4 have an interesting interpretation from the viewpoint of complexity theory. They imply a trade-off between the number of available primitives and the availability of an appropriate amount of a resource that, as we shall presently explain, can be intuitively identified with

temporary storage (hence the term "temporary-storage channel" introduced in Section 3.)

Following the lines of the proof of Theorem 5.3, any invertible function f (Figure 5.4a) can be realized by a network of the kind illustrated in Figure 5.4b (where only some relevant details have been indicated).

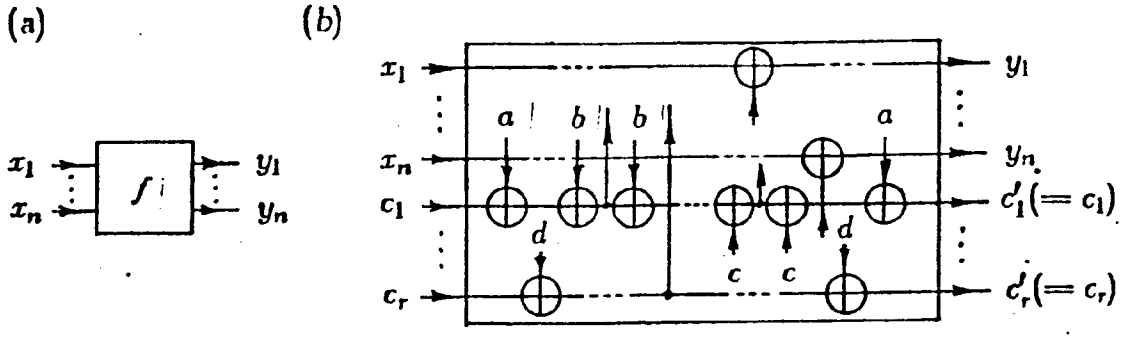


FIG. 5.4 *If instead of an ad hoc reversible mechanism (a) for the invertible function f one seeks a mechanism based on the* AND/NAND *primitive, as in (b), auxiliary constant input signals are required. Such signals are returned unchanged at the output.*

Let us consider the process of traversing the box of Figure 5.4b from left to right. Each line running through the box represents the evolution of a binary storage element. In general, all the constants entering the box will be repeatedly written over as they traverse the box itself; yet, they will emerge from the box with the original values. In this context, a temporary storage channel (such as the pair $(\{c_1, \ldots, c_r\}, \{c'_1, \ldots, c'_r\})$) represents a "scratchpad" register which is initialized to an assigned state (say, all 0's) and is invariably restored to the original state before the end of the computation. (Though more general, this behavior is analogous to that of the reversible Turing machines described by Bennett[4] and briefly discussed in Section 7.)

In this context, Theorem 5.2 can be interpreted as saying that, for a given choice of reversible primitives, it may be impossible to carry out the computation of a given invertible finite function if one is restricted to working on a register of size just enough to contain the argument (cf. the limitations of linear-bounded automata). On the other hand, Theorem 5.3 says that this difficulty disappears as soon as one permits the use of an auxiliary temporary-storage register of appropriate size.

The following list (cf. Figure 5.5) sums up in a schematic way the input/output resources of which a reversible network must avail itself in order