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.

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.*

1. Introduction

This is an abridged version of a much longer report of the same title[27], to which the reader may turn for further details, most proofs, and extended references. Here, the numbering of formulas, figures, etc. reflects that of the original version.

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.

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 microscopic 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.

---

*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.

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 $\phi\colon X \to Y$ 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. 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 $\mathbf{B} = \{\mathbf{0}, \mathbf{1}\}$. A finite function is *of order* n if it has n input lines.

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. Thus, 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 $\langle x \rangle \mapsto \langle x, \ldots, x \rangle$. Intuitively, the responsibility for providing fan-out is shifted from the composition rules to the computing primitives.

Abstract computers (such as finite automata and Turing machines) are essentially function-composition schemes. It is customary to express a function-composition scheme in graphical form as a *causality network*. This is basically an acyclic directed graph in which nodes correspond to functions and arcs to variables. 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 (such as the use of special markers called *delay elements*), causality networks having a particular iterative structure can be represented more compactly as *sequential* networks.

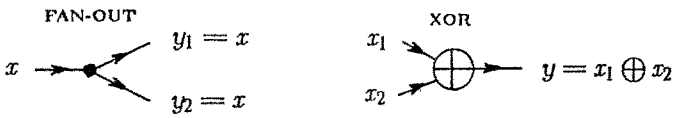
A causality network is *reversible* if it is obtained by composition of invertible primitives. Note that a reversible combinational network always defines an invertible function. Thus, in the case of combinational networks the structural aspect of "reversibility" and the functional aspect of "invertibility" 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.

We shall assume familiarity with the concept of "realization" of finite functions 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.

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

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



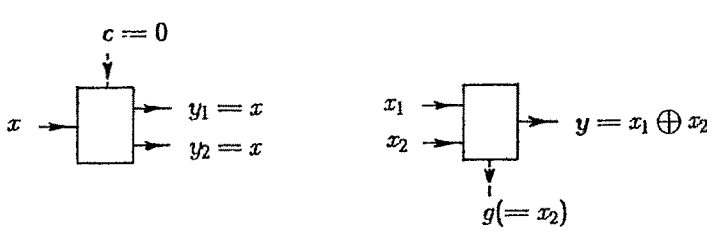
Neither of these functions is invertible. (Indeed, FAN-OUT is not *surjective*, since, for instance, the output $\langle 0, 1\rangle$ cannot be obtained for any input value; and XOR is not *injective*, since, for instance, the output 0 can be obtained from two distinct input values, $\langle 0, 0\rangle$ and $\langle 1, 1\rangle$). Yet, both functions admit of an invertible realization.

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

$$\begin{array}{cc} 0\,0 & 0\,0 \\ 0\,1 & 1\,1 \\ 1\,0 & 1\,0 \\ 1\,1 & 0\,1 \end{array}, \tag{3.2}$$

which we have copied over with different headings in (3.3a), (3.3b), and (3.6b). Then, FAN-OUT can be realized by means of this function* as in (3.3a) (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 (3.3b), by simply disregarding the auxiliary output component g. In more technical terms, (3.1a) is obtained from (3.3a) by componentwise *restriction*, and (3.1b) from (3.3b) by *projection*.

$$\text{(a)}\quad \begin{array}{ccc} c & x & y_1\,y_2 \\ 0 & \boxed{0} & \boxed{0}\,\boxed{0} \\ 0 & \boxed{1} & \boxed{1}\,\boxed{1} \\ 1 & 0 & 1\;0 \\ 1 & 1 & 0\;1 \end{array} \qquad \text{(b)}\quad \begin{array}{cc} x_1\,x_2 & y\;g \\ \boxed{0}\,\boxed{0} & \boxed{0}\,0 \\ \boxed{0}\,\boxed{1} & \boxed{1}\,1 \\ \boxed{1}\,\boxed{0} & \boxed{1}\,0 \\ \boxed{1}\,\boxed{1} & \boxed{0}\,1 \end{array} \tag{3.3}$$

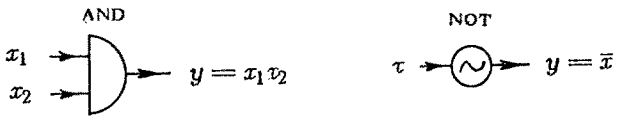


*Ordinarily, one speaks of a realization "by a network." Note, though, that a finite function by itself constitutes a trivial case of combinational network.

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 (3.3a), and *the sink* the auxiliary output components such as g in (3.3b). 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 to construct an invertible realization of a given function.

$$\text{(a)}\quad \begin{array}{cc} x_1 & x_2 \\ 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{array} \rightarrow \begin{array}{c} y \\ 0 \\ 0 \\ 0 \\ 1 \end{array} \qquad \text{(b)}\quad \begin{array}{c} x \\ 0 \\ 1 \end{array} \rightarrow \begin{array}{c} y \\ 1 \\ 0 \end{array} \tag{3.4}$$

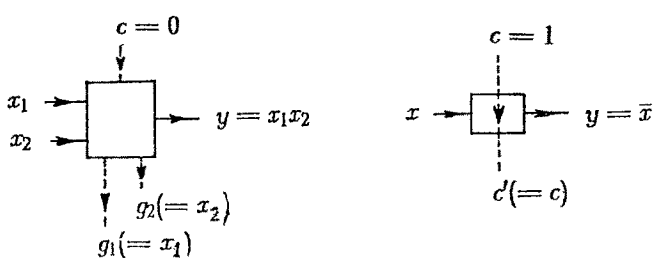


For example, from the *invertible* function AND/NAND defined by the table

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

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

$$\text{(a)}\quad \begin{array}{ccc} c & x_1 & x_2 \\ \boxed{0} & 0 & 0 \\ \boxed{0} & 0 & 1 \\ \boxed{0} & 1 & 0 \\ \boxed{0} & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{array} \rightarrow \begin{array}{ccc} y & g_1 & g_2 \\ \boxed{0} & 0 & 0 \\ \boxed{0} & 0 & 1 \\ \boxed{0} & 1 & 0 \\ \boxed{1} & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{array} \qquad \text{(b)}\quad \begin{array}{cc} x & c \\ 0 & 0 \\ \boxed{0} & 1 \\ 1 & 0 \\ \boxed{1} & 1 \end{array} \rightarrow \begin{array}{cc} y & c' \\ 0 & 0 \\ \boxed{1} & 1 \\ 1 & 0 \\ \boxed{0} & 1 \end{array} \tag{3.6}$$



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 (3.6*a*)—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.

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

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

Thus, 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*.

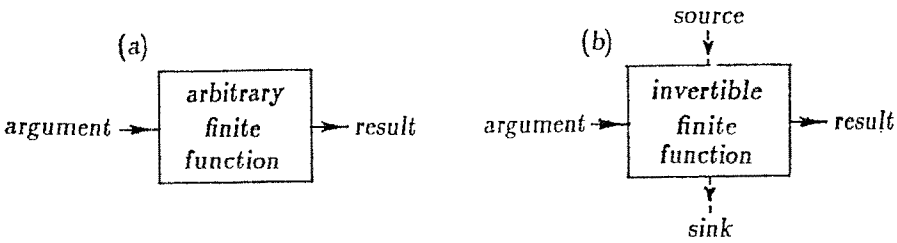


FIG. 4.2 *Any finite function (a) can be realized as an invertible finite function (b) having a number of auxiliary input lines which are fed with constants and a number of auxiliary output lines whose values are disregarded.*

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 of very simple primitives.

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 5.1*c*. Referring to (3.6*a*), observe that $y = x_1x_2$ (AND function) when $c = \mathbf{0}$, and $y = \overline{x_1x_2}$ (NAND function) when $c = \mathbf{1}$. 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 (3.3*a*), observe that $y_1 = y_2 = x$ when $c = \mathbf{0}$ (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 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 $\langle \mathbf{1}, p, q\rangle \mapsto \langle \mathbf{1}, p, p \oplus q\rangle$.)

In the element-by-element substitution procedure outlined above, the number of source and sink lines that are introduced is roughly proportional to the number of computing elements that make up the original network. 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.

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. First, we shall prove that any invertible finite function can be realized *isomorphically* from certain generalized AND/NAND primitives. Then, we shall prove that any of these primitives can be realized from the AND/NAND element possibly with temporary storage but with *no garbage*.

DEFINITION 5.1 Consider the set $\mathbf{B} = \{0,1\}$ with the usual structure of Boolean ring, with "$\oplus$" (*exclusive*-OR) denoting the addition operator, and juxtaposition (AND) the multiplication operator. For any $n > 0$, the *generalized* AND/NAND *function of order* n, denoted by $\theta^{(n)}: \mathbf{B}^n \to \mathbf{B}^n$, is defined by

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

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*.

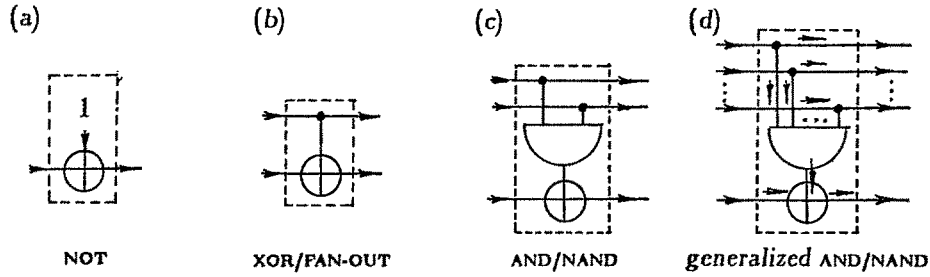


FIG. 5.1 *Graphic representation of the generalized* AND/NAND *functions.* WARNING: This representation is offered only as a mnemonic aid in recalling a function's truth table, and is not meant to imply any "internal structure" for the function, or suggest any particular implementation mechanism. *(a)* $\theta^{(1)}$*, 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.*

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

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.*

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 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) all $\theta^{(i)}$ for $i \leq n$, where n is the order of the given function. 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 $\langle\{c\},\{c'\}\rangle$ constitutes a temporary-storage channel. When $c = \mathbf{0}$, the remaining variables behave as the corresponding ones of $\theta^{(n)}$.∎

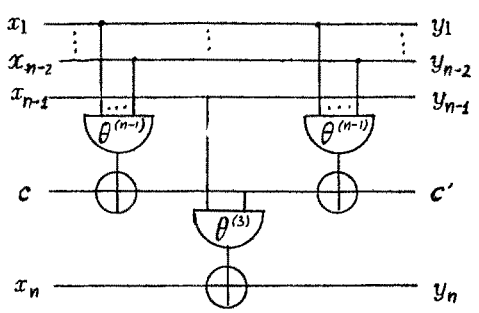


FIG. 5.3 *Realization with temporary storage of $\theta^{(n)}$ from $\theta^{(n-1)}$ (and $\theta^{(3)}$). In this network, when $c = \mathbf{0}$, also $c' = \mathbf{0}$, and the remaining components behave as the corresponding ones of $\theta^{(n)}$.*

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 [7] for more specific examples of destructive interference of garbage.

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 to be able to compute a finite function ϕ.

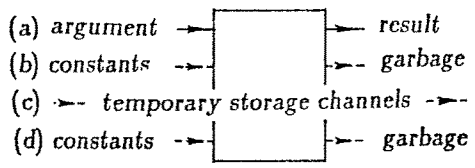


FIG. 5.5 *Classification of input and output lines in a reversible combinational network, according to their function. (a) Argument and result of the intended computation. (b) Constant and garbage lines to account for the noninvertibility of the given function. (c) "Temporary storage" registers required when only a restricted set of primitives is available. (d) Additional constant and garbage lines required when in designing the network one chooses not to take full advantage of the correlation between internal streams of data, and thus looses opportunities to bring about destructive interference of garbage.*

6. Conservative logic

Universal logic capabilities can still be obtained even if one restricts one's attention to combinational networks that, in addition to being reversible, conserve in the output the number of **0**'s and **1**'s that are present at the input. The study of such networks is part of a discipline called *conservative logic* [7] (also cf. [11]). As a matter of fact, most of the results of Sections 4 and 5 were originally derived by Fredkin and associates in the context of conservative logic.

In conservative logic, all data processing is ultimately reduced to *conditional routing* of signals. Roughly speaking, signals are treated as unalterable objects that can be moved around in the course of a computation but never created or destroyed.

The basic primitive of conservative logic is the *Fredkin gate*, defined by the table

c	x_1	x_2		c'	y_1	y_2
0	0	0		0	0	0
0	0	1		0	1	0
0	1	0		0	0	1
0	1	1	→	0	1	1
1	0	0		1	0	0
1	0	1		1	0	1
1	1	0		1	1	0
1	1	1		1	1	1

(6.1)

This computing element can be visualized as a device that performs conditional crossover of two *data* signals a and b according to the value of a *control* signal c (Figure 6.1a). When $c = 1$ the two data signals follow parallel paths, while when $c = 0$ they cross over (Figure 6.1*b*).

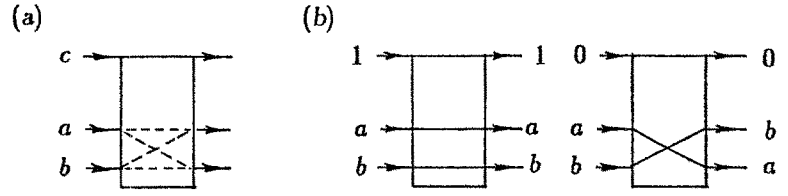


FIG. 6.1 *(a) Symbol and (b) operation of the Fredkin gate.*

In order to prove the universality of this gate as a logic primitive for reversible computing, it is sufficient to observe that AND can be obtained from the mapping $\langle p, q, \mathbf{0}\rangle \mapsto \langle p, pq, \bar{p}q\rangle$, and NOT and FAN-OUT from the mapping $\langle p, \mathbf{1}, \mathbf{0}\rangle \mapsto \langle p, p, \bar{p}\rangle$.

In a conservative logic circuit, the number of 1's, which is conserved in the operation of the circuit, is the sum of the number of 1's in different parts of the circuit. Thus, this quantity is an additive "integral of the motion," and can be shown to play a role analogous to that of energy in physical systems. Other connections between conservative logic and physics will discussed in more detail in [7], where, in particular, we describe a physical realization of the Fredkin gate based on elastic collisions.

7. Reversible sequential computing

In Sections 4 and 5, we started from a certain computing object (viz., a finite function), and we discussed the conditions for its *reversible* realization first (*a*) as an object of the same nature (viz., an invertible finite function) treated as a "lumped" system, thus stressing *functional* aspects, and then (*b*) as a "distributed" system (viz., a reversible combinational network), thus stressing *structural* aspects and paving the way for a natural physical implementation.

By and large, we shall follow a similar plan in dealing with the more complex computing objects that constitute the paradigms of sequential computing, namely, *finite automata* (in the present section), *Turing machines* (Section 8), and *cellular automata* (Section 9).

By definition, a finite automaton is *reversible* if its transition function is invertible. Thus, in order to realize a finite automaton by means of a reversible sequential network, it will be sufficient to take its transition function, construct a reversible realization of it, and use this as the combinational part of the desired sequential network. The problem of reversibly realizing an arbitrary finite function has been solved in Section 4. Thus, we have the following theorem.

THEOREM 7.1. *For every finite automaton* $\tau: X \times Q \to Q \times Y$, *where* $X = B^m$, $Y = B^n$, *and* $Q = B^u$, *there exists a reversible finite automaton* $t: (B^r \times B^m) \times B^u \to B^u \times (B^n \times B^{r+m-n})$, *with* $r \leq n + u$, *such that*

$$t_i(\overbrace{\mathbf{0}, \ldots, \mathbf{0}}^{r}, x_1, \ldots, x_m, q_1, \ldots, q_u) = \tau_i(x_1, \ldots, x_m, q_1, \ldots, q_u), \quad (i = 1, \ldots, u + n).$$

In other words, whatever can be computed by an arbitrary finite automaton according to the scheme of Figure 7.3a can also be computed by a *reversible* finite automaton according to the schema of Figure 7.3*b*.

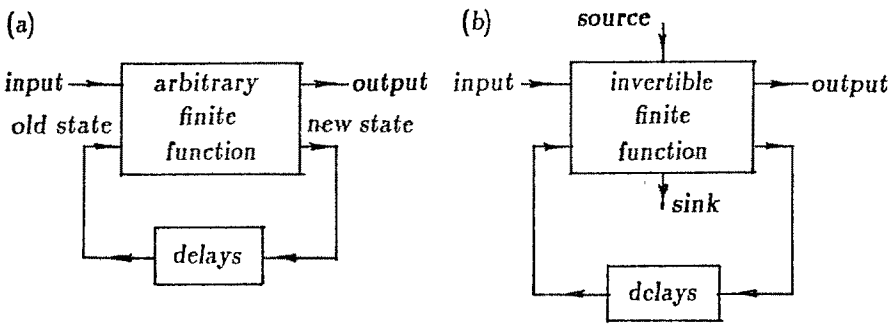


FIG. 7.3 *Any finite automaton (a) can be realized as a reversible finite automaton (b) having a number of auxiliary input lines (source) which are fed with constants and a number of auxiliary output lines (sink) whose values are disregarded.*

Having discussed the realization of finite automata by means of reversible finite automata, we turn now to the realization of finite automata by means of reversible finite sequential networks based on given primitives.

It is clear that all the arguments of Sections 5 and 6 concerning finite functions immediately apply to the transition function of any given finite automaton. In particular, every finite automaton can be realized by a finite, reversible sequential network based on, say, the AND/NAND primitive. With reference to Figure 5.5, one can visualize such a realization by feeding back, via delay elements, some of the result lines to some of the argument lines, and *all* all of the temporary-storage outputs to the corresponding inputs. In order to insure the desired behavior, the delay elements associated with the temporary-storage channels must be initialized *once and for all* with appropriate values (typically, all **0**'s), while the source lines must be fed with appropriate constants (typically, all **0**'s) *at every sequential step.*

In a conventional computer, power dissipation is proportional to the number of logic gates. On the other hand, the number of constants/garbage lines in Figure 7.4 is at worst proportional to the number of input/output lines (cf. Theorems 4.1 and 5.3). From the viewpoint of a physical implementation, where signals are encoded in some form of energy, the above schema can be interpreted as follows: *Using invertible logic gates, it is ideally possible to build a sequential computer with zero internal power dissipation.* Power dissipation might arise outside the circuit, typically at the input/output interface, if the user chose to connect input or output lines to nonreversible digital circuitry. Even in this case, power dissipation is at most proportional to the number of argument/result lines,* rather than to the number of logic gates (as in ordinary computers), and is thus independent of the "complexity" of the function being computed. This constitutes the central result of the present paper.

8. Reversible Turing machines

We shall assume the reader to be familiar with the concept of *Turing machine.* From

*According to Theorem 4.1, the number of constant lines need not be greater than that of result lines, and the number of garbage lines need not be greater than that of argument lines.

our viewpoint, a Turing machine is a closed, time-discrete dynamical system having three state components, i.e., (a) an infinite *tape*, (*b*) the internal state of a finite automaton called *head*, and (c) a *counter* whose content indicates on which tape square the head will operate next. Let T, H, and C be the sets of tape, head, and counter states, respectively. A Turing machine is *reversible* if its transition function $\tau: T \times H \times C \rightarrow T \times H \times C$ is invertible.

It is well known that for every recursive function there exists a Turing machine that computes it, and, in particular, that there exist computation-universal Turing machines. Are these capabilities preserved if one restricts one's attention to the class of *reversible* Turing machines?

The answer to the above question is positive. In fact, in [4] Bennett exhibits a procedure for constructing, for *any* Turing machine and for certain quite general computation formats, a reversible Turing machine that performs essentially the same computations.

In order to obtain the desired behavior, Bennett's machine is initialized so that all of the tape is blank except for one connected portion representing the computation's argument, and the head is set to a distinguished "initial" state and positioned by the argument's first symbol. At the end of the computation, i.e., when the head enters a distinguished "terminal" state, the result will appear on the tape alongside with the argument, and the rest of the tape will be blank. Thus, a number of tape squares that are initially blank will eventually contain the result. These squares fulfill a role similar to that played by the constants/garbage lines in Section 5, in the sense that they provide a sufficient supply of "predictable" input values (blanks) at the beginning of the computation, and collect the required amount of "random" output values (in this case, a copy of the argument—cf. the first row of (4.2)) at the end of the computation. Moreover, during the computation a number of originally blank tape squares may be written over and eventually erased. These squares fulfill a role similar to that played by the temporary-storage lines in Section 5.

It is clear that, like the constants in the reversible combinational networks of Section 5, the blanks in Bennett's machine play an essential role in the computation, since without their presence one could not achieve *universality* and *reversibility* at the same time. Intuitively, computation in reversible systems requires a *higher degree of "predictability" about the environment's initial conditions* than computation in nonreversible ones.

9. Reversible cellular automata

We shall assume the reader to have some familiarity with the concept of *cellular automaton*—in essence an array of identical, uniformly interconnected finite automata[21]. From a physical viewpoint cellular automata are in many respects more satisfactory models of computing processes than Turing machines[22], and for this reason the question of whether there exist reversible cellular automata that are computation- and construction-universal is of particular interest (and has been long debated).

The answer to the above question is positive. In fact, in [20] Toffoli exhibits a procedure for constructing, for *any* cellular automaton (presented as an infinite, space-iterative sequential network), a reversible cellular automaton that realizes it. As in the case of reversible Turing machines, also in reversible cellular automata the predictability of a computing structure's environment plays an essential role in making the computation proceed as intended.

It is well known that any Turing machine can be embedded in a suitable cellular

automaton. Thus, according to the foregoing discussion, any Turing machine can be realized by an infinite *reversible* sequential nework.

10. Conclusions

We have shown that the choice to use reversible mechanisms in describing functional and structural aspects of computing processes is a viable one. What can be gained from this choice?

In the synthesis of an abstract computing system, the requirement that the system be reversible can in general be met only at the cost of greater structural complexity. This "logical" overhead is quite slight; on the other hand, the system's very reversibility promises to be a key factor in leading to a more efficient physical realization, since, at the microscopic level, the "primitives" and the "composition rules" available in the physical world resemble much more closely those used in the theory of reversible computing than those used in traditional logic design.

Acknowledgments

Many ideas discussed in the present paper were originated by Prof. Edward Fredkin, to whom I also owe much useful advice and encouragment.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-75-C-0661.

List of references

[4] Bennett, C. H., "Logical Reversibility of Computation," *IBM J. Res. Dev.* **6** (1973), 525-532.

[7] Fredkin, Edward, *and* Toffoli, Tommaso, "Conservative Logic," (in preparation). Some of the material of this paper is tentatively available in the form of unpublished notes from Prof. Fredkin's lectures, collected and organized by Bill Silver in a *6.895 Term Paper*, "Conservative Logic," and in the form of another *6.895 Term Paper*, "A Reversible Computer Using Conservative Logic," by Edward Barton, both at the MIT Dept. of Electr. Eng. Comp. Sci. (1978).

[11] Kinoshita, Kozo, *et al.*, "On Magnetic Bubble Circuits," *IEEE Trans. Computers* **C-25** (1976), 247-253.

[12] Landauer, Rolf, "Irreversibility and Heat Generation in the Computing Process," *IBM J.* **5** (1961), 183-191.

[20] Toffoli, Tommaso, "Computation and Construction Universality of Reversible Cellular Automata," *J. Comput. Syst. Sci.* **15** (1977), 213-231.

[21] Toffoli, Tommaso, "Cellular Automata Mechanics" (Ph. D. Thesis), *Tech. Rep. no. 208*, Logic of Computers Group, Univ. of Michigan (1977).

[22] Toffoli, Tommaso, "The Role of the Observer in Uniform Systems," *Applied General Systems Research* (ed. G. J. Klir), 395–400 (Plenum Press, 1978).

[23] Toffoli, Tommaso, "Bicontinuous Extensions of Invertible Combinatorial Functions," *Tech. Memo MIT/LCS/TM-124*, MIT Lab. for Comp. Sci. (1979) (to appear in *Math. Syst. Theory*).

[27] Toffoli, Tommaso, "Reversible Computing," *Tech. Memo MIT/LCS/TM-151*, MIT Lab. for Comp. Sci. (1980).