Physics and Computation

by Norman H. Margolus

Submitted to the Department of Physics in partial
fulfillment of the requirements for the degree of

Doctor of Philosophy in Physics

at the

Massachusetts Institute of Technology

June, 1987

Signature of Author ______

Department of Physics
May, 1987

Certified by ______

Edward Fredkin
'hesis Supervisor

Accepted by ______

George F. Koster
Chairman, Graduate Committee

Physics and Computation
by
Norman H. Margolus

Submitted to the Department of Physics on May 15, 1987 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Physics.

Abstract

Physics imposes fundamental constraints on the ultimate potentialities of computing mechanisms.

The most prominent fundamental constraint coming from physics that is felt today is the finiteness of the speed of light. This constraint implies that communication paths inside of a computer should be as short as possible. For maximum speed, we would also like to have massive parallelism. This motivates us to consider the computational capabilities of cellular automata: uniform arrays of identical processors, each communicating only with nearby neighboring processors.

Another constraint concerns heat dissipation, which limits the maximum size and density of computers. Just as reversible engines are ideally the most energy efficient engines, logically reversible computations (which can be implemented in terms of thermodynamically reversible mechanisms) are ideally the most energy efficient. This motivates us to consider the computational capabilities of reversible logic.

Before one can contemplate actually building computers based on reversible logic and cellular automata, it is necessary to demonstrate that computation is possible in such systems. The compatibility of computation with cellular automata was first demonstrated by von Neumann; theoretical objections concerning the compatibility of computation with reversibility were first answered by Bennett. Toffoli dealt with the combination of reversibility and cellular automata, but not in a way that could make use of Bennett's technique for making reversible computations practicable. The first cellular automata models which incorporate reversibility in a way which makes computation practicable are given here.

Constraints arising from quantum mechanics will presumably be felt as computer elements continue to get smaller. Benioff was the first to address the question of whether or not a microscopic quantum Hamiltonian system can perform exact deterministic computation. He pointed out that any computer for which the time development is generated by the Schrödinger equation must be a reversible computer. Feynman presented the first convincing time-independent quantum Hamiltonian model of computation. Here I use one of my reversible cellular automata models of computation as the basis of explicit quantum Hamiltonian models, and address for the first time the problem of constructing quantum models of parallel computation. I introduce a simple scheme for producing models which simulate a synchronous evolution without any global synchronization, and use this as the basis of a partially successful parallel model, which points up certain difficulties.

Another major facet of the research presented here deals with computers optimized for the simulation of cellular automata. The reversibility and quantum mechanical issues are rather far from limiting current computers, but even in the context of current technology there are enormous advantages in terms of speed, simulation size, and cost that are available to machines tailored specifically for cellular automata. Toffoli and I designed the first general purpose cellular automata machine for use in investigating some of the theoretical models we had constructed. This machine has had a significant impact on the advent of new physical models based on cellular automata (such as the recent lattice gas models of fluid dynamics), and we have arranged for a commercial version to be made available to investigators. I discuss here the architecture and use of this latest version, as well as give a design for the first cellular automata machine that will be able to perform massive 3-dimensional simulations (it will have billions of computational degrees of freedom, each of which will be updated 100 times per second).

Finally, a large number of original results are presented here concerning reversible logic, reversible computation, the construction of reversible cellular automata, invariants in reversible cellular automata, and the applicability of various ideas from physics to the analysis (and synthesis) of reversible models of computation.

Thesis Supervisor: Edward Fredkin

Contents

Acknowledgements

I would first of all like to thank Edward Fredkin, my thesis advisor, without whose support and encouragement I would have found it very difficult to write a thesis on the present topic. His ideas on using theoretical physics as a source of new computational principles and constraints, and his invention of reversible models of computation form the theoretical and motivational bases of this work.

I would next like to thank Tommaso Toffoli, who has been a fellow swimmer in these unconventional waters, and a collaborator on many projects. Together we developed the CAM machines—windows onto cellular automata worlds. Numerous discussions and arguments with him have contributed greatly to my understanding of the field of Information Mechanics. I thank him also for his encouragement in finally escaping from other projects and getting this thesis completed.

Next, Charles Bennett, one of the co-discoverers of reversible computation (along with Fredkin and Toffoli). He has been one of the principal users of our Cellular Automata Machines, and has shared his deep understanding of issues relating to thermodynamics and statistical mechanics.

In addition to these good friends, I would like to thank my family for their understanding and support in my rather extended course of research; the families of my co-workers who have taken me into their homes as one of them; and all of my other friends who have helped keep me sane.

Finally, I would also like to thank co-workers in the field who have helped me in my research—particularly Richard Feynman whom I enormously enjoyed visiting for several months at Cal Tech, as well as P. A. Benioff, U. Frisch, Y. Pomeau, G. Y. Vichniac (who was until recently

part of our research group), and S. Wolfram—as well as acknowledge practical support and encouragement from my friends and co-workers at the MIT Laboratory for Computer Science, and from my thesis committee (Professors John Joannopoulos and Felix Villars).

This research was supported in part by the following government agencies: Defense Advanced Research Projects Agency (Grant No. N00014-83-10-0125), National Science Foundation (Grant No. 8214312-IST), U.S. Department of Energy (Grant No. DE-AC02-83-ER13082); and by a grant from International Business Machines (Grant No. 3260).

Norman Margolus

May 15, 1987

Physics and Computation

Introduction

The ultimate potentialities and limitations of computing mechanisms depend upon the efficiency with which physical interactions can be used to perform computations. This in turn depends upon what we call a computation, and to what extent our conceptual models of computation can be restructured to make them more compatible with the resources offered by nature.

0.1 What is a computation?

All of the conceptual models of computation that we will deal with will be based on the idea of *digital* computation—computation performed in terms of state variables that have a discrete and finite spectrum of possible values. While it is true that the dynamical evolution of a non-digital system such as a wind tunnel can perform a kind of computation for us, digital computation has properties which make it so attractive that it has become virtually synonymous with the word 'computation.'

First of all, digital computation is an abstraction of the way that we use a finite set of discrete symbols to deal with arbitrary information. All possible mechanical manipulations of such symbolic data can be performed by a digital computer.

Secondly, digital computation is exact. We can build computers that have an arbitrarily small chance of having made an error in the course of billions of billions of operations. This leads to what is perhaps the paradox of computation: our ability to predict the details of a computer's operation is so great that we entrust to it computations the outcomes of which are completely surprising.

Finally, all physically reasonable formalizations of the process of *mechanically manipulating symbols* that have been thought of have been shown to be equivalent: they permit exactly the same set of computations to be performed. This is a consequence of the fact that if a model system is general purpose enough to be considered a computer, then it can manipulate a symbolic description of any other computer, and exactly simulate its operation. In this sense all computers are logically equivalent. The only difference between the logical capabilities of a personal computer and a supercomputer (or any of the other models of computation that we will deal with here) is quantitative: given enough time and memory, any computer can perform any computation that any other computer can.

Thus a computation is any sequence of mechanical manipulations of symbolic data that a computer can perform—it doesn't matter which computer, since they all perform the same set of computations.

0.2 Compatibility with nature

Most current conceptual models of computation ignore important general properties of physical systems. This research deals with finding and studying new conceptual models of computation which are more compatible with fundamental physical constraints on computing mechanisms.

The most prominent fundamental constraint coming from physics that is felt today is the finiteness of the speed of light. This constraint implies that communication paths inside of a computer should be as short as possible. For maximum speed, we would also like to have massive parallelism. This motivates us to consider the computational capabilities of cellular automata: uniform arrays of identical processors, each communicating only with nearby neighboring processors.

Another constraint concerns heat dissipation, which limits the maximum size and density of computers. Just as reversible engines are ideally the most energy efficient engines, logically reversible computations (which can be implemented in terms of thermodynamically reversible mechanisms) are ideally the most energy efficient. This motivates us to consider the computational capabilities of reversible logic.

Before one can contemplate actually building computers based on reversible logic and cellular automata, it is necessary to demonstrate that computation is possible in such systems. The compatibility of computation with cellular automata was first demonstrated by von Neumann; theoretical objections concerning the compatibility of computation with reversibility were first answered by Bennett. Toffoli dealt with the combination of reversibility and cellular automata, but not in a way that could make use of Bennett's technique for making reversible computations practicable. The first cellular automata models which incorporate reversibility in a way which makes computation practicable are given here.

Constraints arising from quantum mechanics will presumably be felt as computer elements continue to get smaller. Benioff was the first to address the question of whether or not a microscopic quantum Hamiltonian system can perform exact deterministic computation. He pointed out that any computer for which the time development is generated by the Schrödinger equation must be a reversible computer. Feynman presented the first convincing time-independent quantum Hamiltonian model of computation. Here I use one of my reversible cellular automata models of computation as the basis of explicit quantum Hamiltonian models, and address for the first time the problem of constructing quantum models of parallel computation. I introduce a simple scheme for producing models which simulate a synchronous evolution without any global synchronization, and use this as the basis of a partially successful parallel model, which points up certain difficulties.

Another major facet of the research presented here deals with computers optimized for the simulation of cellular automata. The reversibility and quantum mechanical issues are rather far from limiting current computers, but even in the context of current technology there are enormous advantages in terms of speed, simulation size, and cost that are available to machines tailored specifically for cellular automata. Toffoli and I designed the first general purpose cellular automata machine for use in investigating some of the theoretical models we had constructed. This machine has had a significant impact on the advent of new physical models based on cellular automata (such as the recent lattice gas models of fluid dynamics), and we have arranged for a commercial version to be made available to investigators. I discuss here the architecture and use of this latest version, as well as give a design for the first cellular automata machine that will be able to perform massive 3-dimensional simulations (it will have billions of computational degrees of freedom, each of which will be updated 100 times per second).

Finally, a large number of original results are presented here concerning reversible logic, reversible computation, the construction of reversible cellular automata, invariants in reversible cellular automata, and the applicability of various ideas from physics to the analysis (and synthesis) of reversible models of computation.

Chapter 1

Reversible logic

In this chapter, I discuss the need for reversible logic, and describe the reversible models of computation due to Bennett, Fredkin, and Toffoli. In the course of giving this background, I give a new result in Section 1.2.5 about the computational capabilities of conservative logic gates.

I then go on to discuss two original results: the first (Section 1.4) has an important bearing on the role of energy in certain models of computation; the second (Section 1.5) shows how, in an appropriate reversible computational context, a close analogue of the usual thermodynamic argument concerning the maximum efficiency of a heat engine can be made.

1.1 Irreversibility and heat generation

Thermodynamic questions concerning the need for energy dissipation in a computation were first convincingly addressed by Rolf Landauer[40] in the 1960's. What he realized is that logical irreversibility in a computation must ultimately be translated into physical irreversibility in the mechanism which performs the computation. Erasing a *bit* of information in a computation means that some two-state system which is part of the computer must be set to its 'cleared' state, without regard for which of the two possible states it started in. Since the underlying microscopic dynamics is presumed to be strictly reversible, the information which disappears from the computational degrees of freedom when we erase this bit must be transferred to other degrees of freedom. In other words, we must double the number of states available to the non-computational degrees of freedom, for example by transferring $kT \ln 2$ of work into a heat bath at temperature T. This then is the thermodynamic cost of erasing a bit in a computation.

Landauer went on to point out that the elementary acts of computation, such as AND and OR, are irreversible functions—there isn't enough information in the result to reconstruct what the arguments were. Both of these operations entail erasing more than a *bit* of information, since they each involve two inputs and a single output. Thus he concluded that computations constructed out of such irreversible primitives (which were the only kind known at the time) must entail an unavoidable minimum energy dissipation of the order of kT per elementary logical operation.

While this is many orders of magnitude less than the actual dissipation of current devices, technology is advancing rapidly. This seems to present a fundamental barrier which will eventually be confronted.

1.2 Reversible computation

Independently, Bennett, Fredkin, and Toffoli[7,25,71], for a variety of reasons, addressed the question of the necessity of irreversible operations in a computation. Their results have great relevance to our current discussion: a clear demonstration that reversible computation was practicable would eliminate the only known fundamental theoretical barrier to dissipationless computation.

Clearly any computation can be made to run backwards if we simply keep a complete history of all past states. Toffoli used essentially this approach to give a detailed example[72] of a reversible system which can compute—in the process he gave an important counterexample to a purported "proof" that cellular automata models of computation (see Chapter 2) cannot be both universal and reversible[1].[1] This issue will be very important in our discussion of physics-like models of computation, but Toffoli's method of proof doesn't help us much in our present discussion. All he did was to delay the moment when information is

[1] Although Toffoli's systems were only shown to be universal computers as long as the memory space used for recording the history is initially empty, his systems obey a dynamics that is invertible when started from *any* initial state.

erased until the time when the computer is cleared in preparation for the next computation. He didn't show that the amount of information that needs to be erased (and hence the associated unavoidable minimum energy dissipation) could be reduced—this was first shown by Charles Bennett.

Bennett described a reversible Turing machine. A Turing machine[82] is a kind of universal computer traditionally used in theoretical proofs about computability. It is a particular abstraction of the idea of a mechanical computation invented by Alan Turing—it formalizes the idea of what an unintelligent person could compute by mechanically following instructions, and predates the advent of the electronic digital computer. A Turing machine has two parts, called a *head* and a *tape*. The tape is a one-dimensional array of *cells*, each of which can hold a single symbol, chosen from a finite alphabet of symbols. The tape is taken to be unbounded (to eliminate considerations stemming from a particular memory size) and is always initially blank, except for a finite region. The head is a movable mechanism, and is always positioned at some particular cell of the tape. It can be in one of a finite number of states, and based only on its current state and the symbol contained in the cell it is at, it obeys a rule which tells it whether to write a new symbol, and whether to move one position either right or left. Turing machines are known to be universal computers—they are logically equivalent to any other general purpose computer in that they can exactly simulate the operation of any other computer (all universal computers can perform the same set of computations, given enough time and memory).

Bennett's Turing machine operated reversibly at every step—it had an extra tape used for keeping a history of the computation.[2] Bennett realized, though, that at the end of the computation the answer could be copied somewhere, and then the entire computation which produced this answer could be run backwards, *and the history tape would be restored to its initial empty state without performing any irreversible operations* (this important insight will be discussed in more detail in Section 1.2.4). Thus the overall computation took about twice as long as an irreversible version would have, and resulted in leaving everything exactly as it started, except for a copy of the answer—no unmanageable accumulation of useless historical *garbage* occurred. Bennett's mechanism didn't just delay irreversible operations until the moment when the history tape was cleared—it avoided the logical need for irreversible operations (and the associated unavoidable dissipation) altogether.[3] By putting the constraint of reversibility (which comes from microscopic physics) explicitly into his model, he could reorganize the way the com-

[2] Like Toffoli's machines (which, incidentally, were conceived several years after Bennett's) this machine operates invertibly started from *any* initial state, although it needs to start with a clean history tape (or at least a history tape started in some standard state) to perform a computation.

[3] Except for any irreversibility involved in changing the program in preparation for the next computation—the number of irreversible operations needed to do this is at worst proportional to the length of the program, but independent of the length of the computation.

putation was done in order to "erase" historical *garbage* reversibly.[4]

Turing machines provide a certain abstraction of physical computability—if a computation can be done by a Turing machine, it can be done by an actual physical machine. Another abstraction that is commonly used by electronic engineers is that of logic gates. Knowing the rules for composing logic elements, machines can be designed which can be expected to work. The logic gate comes closer than the Turing machine to capturing realistic aspects of physics, such as the fact that computation can be going on at more than one place at a time, and that gates must be close to each other if a signal is to travel between them in a very short time. Thus we expect that an analysis of computation within the context of reversible logic will provide a more fundamental insight into issues of computational efficiency. For this reason I regard Fredkin and Toffoli's demonstration of the ability of reversible logic elements to perform computations to be of particular importance. Before we can discuss this in detail, however, we must make a brief digression into the subject of logic elements.

[4] In [8], Bennett discusses a version of such a reversible Turing machine which operates in a manner analogous to the genetic mechanism for RNA synthesis. Such a computer is reversible in the same way that chemical reactions are—the direction of operation is determined by the relative concentrations of reactants and products. If a very small bias away from equilibrium concentrations is maintained in certain reactant and product molecules, his computer would take almost as many backward steps as forward steps, and could in principle operate with arbitrarily little dissipation: the total dissipation for a given computation is proportional to the average rate at which the computation drifts forwards.

1.2.1 Universal sets of logic gates

First we need a definition: a set of binary logic gates is said to be *universal* if an arbitrary Boolean (i.e., binary valued) function can be implemented as a composition of such gates. Some individual gates are universal—for example, the NAND gate, which implements the following truth table

p	q	p NAND q
0	0	1
0	1	1
1	0	1
1	1	0

This gate returns a 1 unless both inputs are 1's. With one input fixed at a constant (unchanging) value of 1, this gate gives the NOT function—a 1 applied to the free input becomes a 0, and a 0 becomes a 1. If we use a NOT to complement the result of a NAND, the function realized by this composition is the AND function, and has the following truth table

p	q	p AND q
0	0	0
0	1	0
1	0	0
1	1	1

If instead we use NOT to complement both inputs to a NAND, the composite function is the OR function, and has this truth table

p	q	p OR q
0	0	0
0	1	1
1	0	1
1	1	1

Notice that AND gives the same result as ordinary multiplication of binary quantities, while OR is the same as addition, except in the 11 case—it is traditional to adopt the notation used for multiplication and addition when writing functions involving AND and OR.

Given an arbitrary set of Boolean input variables $(a_1, a_2, \ldots, a_n)$, we can write a product term that has a value of 1 only if a given pattern of 1's and 0's appears in the inputs. For example, suppose there are five input variables, and we want to distinguish the case $(a_1, a_2, a_3, a_4, a_5) = (1, 1, 0, 1, 0)$ from all others. The product

$$a_1 a_2 \bar{a}_3 a_4 \bar{a}_5$$

(where the complement of a binary quantity a is written $\bar{a}$) would have a value of 1 in the desired case, and 0 otherwise. If we take an arbitrary truth table and construct a product term corresponding to each 1 in the result column, and then add them together, we have constructed the desired function. Since each product term picks out a different case, no more than one term will give a 1, and so addition can be replaced by

OR. Thus any Boolean function of any number of inputs can be written as a composition of AND, OR, and NOT, and hence as a composition of NAND gates.

In this discussion we have glossed over a point which will be very important when we move on to the discussion of reversible logic: we have implicitly assumed that the output of one logic gate can be connected to the inputs of several other gates. This splitting of signals is referred to as *fanout.* Real gates have this property, although only a specified amount of fanout is available before additional gates need to be used just to get more fanout. In our discussion of invertible logic, fanout will have to be considered carefully: if we want to allow fanout to occur by having a single wire split into two, then we must also allow the inverse operation, where two wires join into one. For this reason, we will not allow wires to split.

The kinds of circuits we have considered in this section, where the output is a function of input variables alone (without any feedback) are referred to as *combinational.* To turn combinational logic into the more general circuitry needed to implement a universal computer, we need to employ a delay element (something for which the output at time $t+1$ equals its input at time t) and allow outputs to be fed back and connected as inputs. If our NAND gate is considered to always have such a delay attached to its output, then NAND is sufficient for constructing general purpose computers (in fact a line of commercial computers was actually built that made use only of NAND gates for its logic).

1.2.2 Universal sets of reversible gates

We begin with Landauer's observation that traditional logic elements are irreversible. One obvious problem is that they have two inputs and one output: no gate which has fewer outputs than inputs can be invertible (assuming both input and output values are taken from the same state set, and there are no constraints on allowed input combinations). This leads us to consider gates which have equal numbers of inputs and outputs. With two inputs, two outputs, and binary variables, there are 256 possible truth tables. Since a reversible function must map each distinct input case onto a distinct output case, each reversible function with two inputs is a permutation which maps a set of four elements

$$(0,0) \quad (0,1) \quad (1,0) \quad (1,1)$$

onto itself. Thus only 24 of the 256 possible truth tables represent invertible functions. If you enumerate all invertible 2-input/2-output functions of Boolean variables which can be constructed using only a composition of XOR gates (gates which return the sum, modulo 2, of their two inputs), you find that there are 24—thus these are all of the invertible functions we are considering. But any combination of XOR gates is a linear function of input variables. For example, a typical output of such a function might be given by

$$a_1 \oplus a_5 \oplus a_9 \oplus 1$$

where $\oplus$ denotes addition modulo 2. For any such function, the output either doesn't depend on a given input variable at all, or else the output changes whenever that input is changed while all other inputs are held constant. No such function can reproduce the NAND table (which requires input cases which differ in a single variable to produce the same output) for any pair of its inputs, and so our set of 24 invertible 2-in/2-out gates fails to be universal.

We must therefore widen our search for invertible logic gates. We can either consider gates with more inputs (and outputs), or gates with more states. With three inputs and using Boolean logic, there are eight possible input cases; with two inputs and using ternary (3 state) logic, there are nine possible input cases. Here we will only look at the Boolean case, which is the one that Fredkin and Toffoli considered.[5]

1.2.3 The Toffoli gate

Once we consider gates with three inputs, there are many gates which are universal by themselves, just as the NAND was in the domain of irreversible logic. A simple example is Toffoli's AND/NAND gate, which

[5] When considering logic with more then two states, it is sufficient to discuss the construction of Boolean functions in order to demonstrate universal computing capability—the third state may be used within circuits which use only two of the three states for inputs and outputs. Using this idea, very simple circuits based on appropriate reversible ternary logic gates with two inputs can be used to simulate universal reversible binary gates with three inputs, and hence prove the universality of the ternary gates.

has this truth table:

p	q	r	p'	q'	r'
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

In this gate, p and q go through unchanged, while the AND of these inputs is XORed (added modulo 2) to r to produce the third output. This is invertible, and is in fact its own inverse (if you connect p', q' and r' to the corresponding inputs of a second gate, the overall function will be the identity function). By holding r fixed at 1 you get the NAND function of p and q at r', and by connecting constants of 1 to q and 0 to r, you find that both p' and r' are copies of p, and so this gate can provide fanout. Timing delays can be associated either with the gates, the wires, or both—in any case we conclude that Toffoli's gate is a universal logic element.

1.2.4 Conservative logic

Fredkin's approach to the same problem was this: he noted that he had many possible choices for a 3-in/3-out invertible logic gate, and so he decided to add another constraint which was intended to make it easier to find a physical realization of his gates. He required that in every case the number of 1's in the output must equal the number in the input, and called the resulting kind of logic *conservative logic.* With three inputs there are 36 distinct invertible gates of this sort: there is no choice for the cases where the input contains three 0's or three 1's; there are $3! = 6$ ways to permute the three cases containing a single 1 and similarly six for the single 0 cases. If, however, we don't distinguish between gates that are equivalent up to a relabeling of the input and output signals, then there are actually only three distinct gates that are possible. With the inputs and outputs named as above, these are:

- the *identity gate*, for which $p' = p$, $q' = q$ and $r' = r$,

- the *Fredkin gate*, for which $p' = p$, and the other two signals either go through unchanged (if $p = 1$) or are interchanged (if $p = 0$).

- the *SMP gate* (symmetric majority/parity gate), a gate that cyclicly permutes all inputs one way if the input count is even ($p' = q$, $q' = r$, and $r' = p$), and the opposite way if it is odd.

Both the Fredkin gate and the SMP gate are universal logic elements, but the former is perhaps a little easier to work with, and so we will concentrate on it (the SMP gate will be mentioned again in

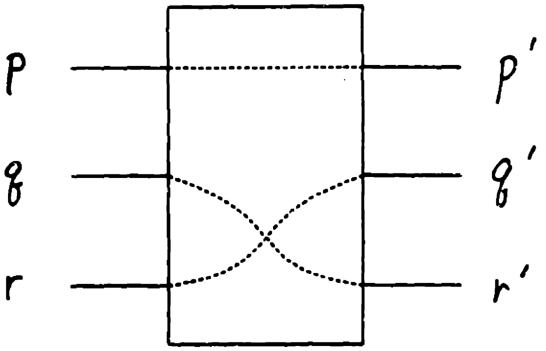


Figure 1.1: The Fredkin Gate. All three inputs go straight through unless p is a zero, in which case $q \rightarrow r'$ and $r \rightarrow q'$.

Section 1.4). Here is the truth table for the Fredkin gate:

p	q	r	p'	q'	r'
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

Graphically, this gate is represented as in Figure 1.1. If r is given a constant (unchanging) value of 0, then r' has a value given by $\bar{p}q$ (i.e., $\bar{p}$ AND q) while q' has a value of pq (i.e., p AND q). If in addition we let q =1, then these become $r' = \bar{p}$ and $q' = p$. Thus we have available the AND, NOT, and FANOUT functions, as we must for this to be a universal

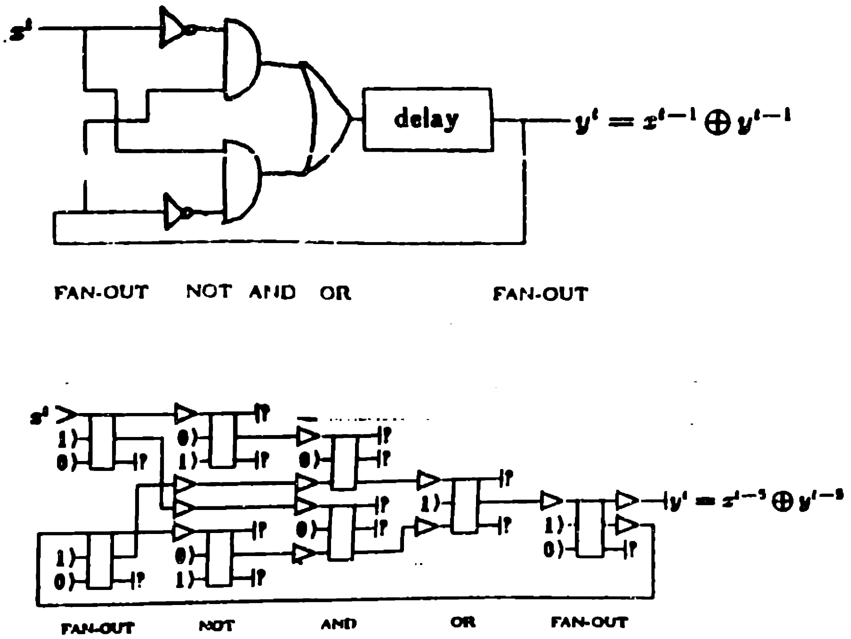


Figure 1.2: Two circuits which both compute the same function. The first is an ordinary sequential circuit made of irreversible gates; the second is a transcription of this circuit into Fredkin gates.

time $t+1$ equals input at time t).

Since we can, in a similar manner, transcribe any conventional circuit into a conservative logic circuit, this type of logic is clearly sufficient to construct computers; but we haven't really arrived at our objective yet. Something must be done with all these *garbage* signals. If we simply erase them, then what we have done is certainly no better than conventional irreversible logic. Instead, we will adapt the idea that Bennett used in his reversible Turing machine, to eliminate essentially all of these extra outputs.

Bennett's argument made use of the notion of an inverse computation. Given any conservative logic circuit, such as the one of Figure1.2, we can construct the inverse circuit by reversing the motion of all signals (i.e., the direction of all arrows), interchanging the roles of inputs and outputs, and converting all gates into inverse gates (since a second permutation undoes the first, the Fredkin gate is its own inverse, just as the Toffoli gate was). If such a 'mirror image' of the circuit of Figure 1.2 is connected to the direct circuit (with appropriate delay elements so that all signals arrive at the inverse circuit at the right moments), then the net effect of the circuit will be to compute the identity function: the input x^t will be reproduced, and all of the constant inputs used to make the Fredkin gate act like an irreversible gate will be reconstituted, and can be fed back to be reused.

So far we have a rather elaborate scheme for doing nothing. But somewhere in the middle of this circuit is the sum modulo 2 of the input—if we add a fanout element at that point, then we have the

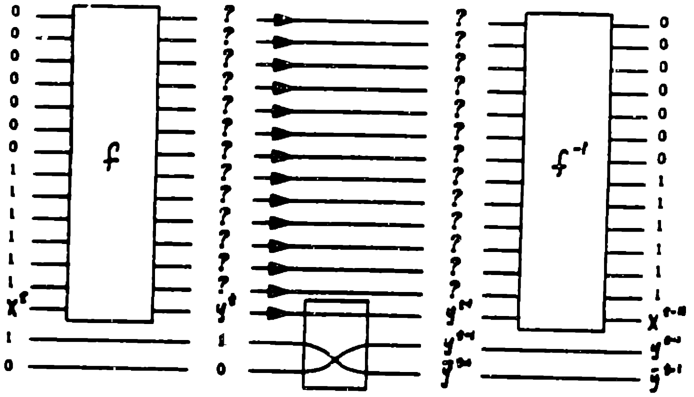


Figure 1.3:

circuit of Figure 1.3. The overall action of this circuit is to convert the input stream x^t plus a constant of 0 and a constant of 1 into a copy of the input x^{t-11} and two copies (one of them complemented) of the 'answer,' $y^{t-1} = x^{t-6} \oplus y^{t-6}$. We get the answer with only one extra step of computation delay compared to the circuit of Figure 1.2—we only needed to copy it. The reconstituted input, however, has the delay associated with both halves of the circuit. Any computation can be treated similarly.[7]

As in Bennett's construction, we were able to reorganize the computation in a way that minimized the production of garbage. Any garbage that is not cleaned up during the computation will eventually have to be erased irreversibly, with a concomitant unavoidable thermodynamic

[7] Likharev, in [43], describes the use of reversible gates based on thermodynamically reversible Josephson junction devices. He concludes that these devices have sufficiently short relaxation times that switching speeds of 10^{-9} seconds can be achieved with a dissipation of $0.01kT$. Ressler[62] gives a logical design for a conservative logic computer.

cost per bit. Thus reversible logic allows us to reduce the amount of unavoidable dissipation from one that is proportional to the number of logical operations (as it always is in a cicuit composed of irreversible logic elements) to one that is independent of the size of the circuit and the lengtn of the computation, and at worst proportional to the number of non-constant inputs.

Given the algorithm we were using in Figure 1.2, our reversible circuit couldn't have gotten the answer any faster, but none of the original constants were reconstituted until we had the answer, and so we needed a maximum number of constants. If we performed partial reversals at intermediate points, the time before we had the answer would be longer, but the computation would need fewer constants at the input.

1.2.5 Other conservative logic gates

With 3 inputs and 3 outputs, we noticed that all (reversible) conservative logic elements which could not be transformed into the identity element by a relabeling of outputs are in fact universal. This property actually holds for conservative Boolean functions with any number of inputs (and outputs), as we will now show.

Consider a conservative logic gate with n inputs labeled a_1, a_2, etc. Let b_1 be the output that is a 1 when only a_1 is a 1 (and all other inputs are 0's), and similarly b_2, etc. (Since each such input must map onto a distinct output case, we can always do this). With this labeling, all gates that are equivalent to the identity gate have $b_1 = a_1$, $b_2 = a_2$, etc., for all possible input cases. We will call all gates that are equivalent to the identity gate under such a labeling *trivial gates*—they could be replaced by separate wires.

If a gate is non-trivial, then for some input there must be a k such that $a_k \neq b_k$. Consider an input case involving the smallest number of 1's for which there is such a k. Because of the conservation of the input count, this input case must have at least two places that differ from the output, one of which is a 1; thus we can without loss of generality assume that for this input case, $a_1 \neq b_1$, and $a_1 = 1$. Our definition of the b's implies that no case involving a single 1 has a difference between input and output, and so we can assume, again without loss of generality, that $a_2 = 1$ in this case. Thus we can write the truth table for a_1 and a_2 (holding all other inputs constant)

a_1	a_2	b_1
0	0	0
0	1	0
1	0	1
1	1	0

Since the case $a_1 = a_2 = 1$ was one involving the fewest 1's for which input and output differ, the first three entries for b_1 must have the values show. This is the truth table for $a_1\bar{a}_2$ (a_1 AND $\bar{a}_2$), which can be used to perform NOT (with a_1 a constant of 1) and AND. To demonstrate fanout, we need only note that in the case $a_1 = a_2 = 1$, since $b_1 = 0$

there must be some output which we'll call b_3 (distinct from b_2) which is a 1, even though a_3 is a 0. Thus

a_1	a_2	a_3	b_1	b_3
1	0	0	1	0
1	1	0	0	1

and so if we let $a_1 = 1$ and $a_3 = 0$, then we get $b_1 = \bar{a}_2$ and $b_3 = a_2$.

Thus we've proven the universality of all non-trivial Boolean conservative logic gates, but we're not quite finished. The construction of Section 1.2.4 that we used to erase *garbage* depends upon the availability of an inverse logic gate; we will show that we can always construct such an inverse gate.

Given a reversible logic gate G, we can form a new gate G^n by connecting together n gates: the outputs of one gate to the corresponding inputs of the next. Since G performs a permutation on the possible input cases, for some power m, G^m is the identity function. Thus G^{m-1} is the inverse of G.

1.3 The Billiard Ball Model

In order to be sure that a computational model is consistent and complete, we would like to be able to find a physical system that, in a suitable idealization, obeys that model. The model that Fredkin found for conservative logic is that of a gas of hard spheres.

The Billiard Ball Model (BBM) is a Classical Mechanical system, and obeys a continuous dynamics—positions and velocities, masses and times are all real variables. In order to make it perform a digital computation, we make use of the fact that integers are also real numbers. By suitably restricting the initial conditions we allow the system to have, and by only looking at the system at regularly spaced time intervals, we can make a continuous dynamics perform a digital process.

In this case, we begin with a 2-dimensional gas of identical hard spheres. If the center of a sphere is present at a given point in space at a given point in time, we will say that there is a 1 there, otherwise there is a 0 there. The 1's can move from place to place, but the total number of 1's never changes.

The key insight behind the BBM is this: *every place where a collision of finite-diameter hard spheres might occur can be viewed as a logic gate.* What path a ball follows depends upon whether or not it hits anything—it makes a decision.

To see how to use this decision to do Boolean logic, consider Figure 1.4. At points A and B and at time t_i, we either put balls at A, B, or both, or we put none. Any balls present are moving as indicated with a speed s. If balls are present at *both* A and B, then they will collide and follow the outer outgoing paths. Otherwise, only the inner outgoing paths will be used.

At time $t = t_i$, position A is a 1 if a ball is there, and 0 otherwise (similarly for position B). At $t = t_f$, the four labeled spots have a ball or no ball—which they have is given by the logical function labeling

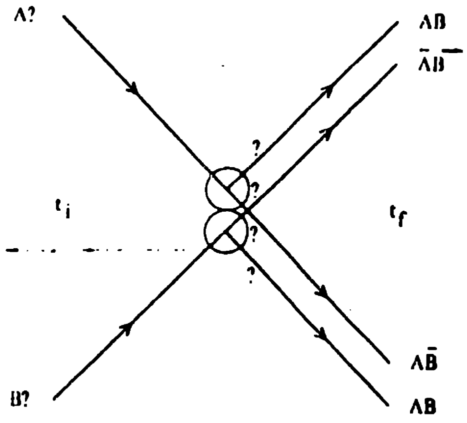


Figure 1.4:

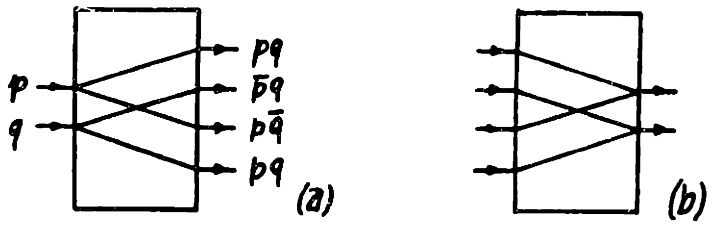


Figure 1.5: Interaction gate. When used backwards, the inputs must be suitably constrained.

the spot. For example, if $A = 1$ and $B = 0$, then the ball coming from A encounters no ball coming from B, and ends up at the point labeled "A and not B".

Thus a place where a collision might occur acts as a reversible, 1-conserving logic-gate, with two inputs and four outputs. Such a collision is shown schematically in Figure 1.5 as a logic gate—called the *interaction gate.* This same schematic symbol can be used with models that involve an attractive interaction (c.f. Section 2.7.1) if the order of the outputs is disregarded.

A path that may or may not contain balls acts as a signal-carrying wire. Mirrors (reflectors) allow bends in the paths. In order to be able to use the outputs from a collision "gate" as inputs to other such gates, we need to very precisely control the angle and timing of the collisions, as well as the relative speeds of the balls. We make this simple to do by severely restricting the allowed initial conditions.

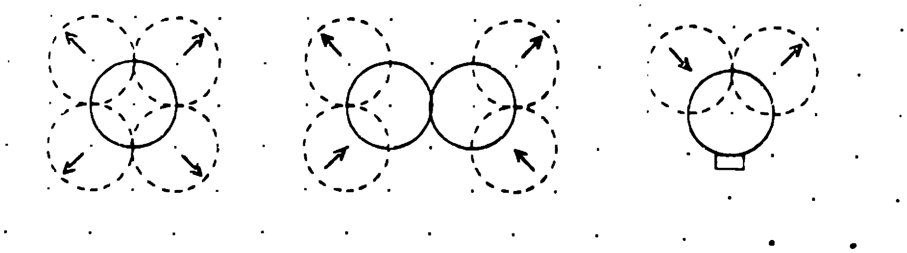


Figure 1.6:

Each ball must start at a grid point of a Cartesian lattice, moving *along* the grid in one of 4 allowed directions. All balls move at the same speed (Figure 1.6a). The time it takes a ball to move from one grid point to another we call our unit of time. The grid spacing is chosen so that balls collide while at grid-points. All collisions are right angle collisions, so that one time-step after a collision, balls are still on the grid (Figure 1.6b). Fixed mirrors are positioned so that balls hit them while at a grid point, and so stay on the grid (Figure 1.6c). By using mirrors, signals can be routed and delayed as required to perform digital logic.

The configuration of mirrors shown in Figure 1.7 solves the problem of making two signals cross without affecting each other. (Notice that if two balls come in together, the signals cross but the balls don't!)

Mirrors and collisions determine the possible paths that signals may follow (*wires*). In order to ensure that all collisions will be right-angle collisions (and not head-on, for example, which would take us off our

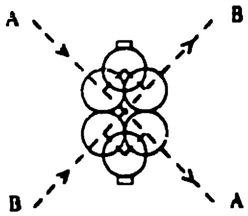


Figure 1.7:

grid} we can label all *wires* with arrows, and restrict initial conditions and interconnections so that a ball found on a given *wire* always moves in the labeled direction.

Thus our universal gates can be connected as required to *build* a computer. Computations can be pipelined—an efficient *assembly-line* way of doing things, where questions flow in one end and finished products (answers) flow out the other, while all the stages in between are kept busy.

Figure 1.8 shows (schematically) the Fredkin gate built out of six interaction gates, three of which are used backwards. Trivial crossovers, in which the logic of the situation ensures that the two paths will never be simultaneously occupied, are indicated by wires that simply cross—the bridge symbol at a crossover point indicates that explicit provisions for a crossover must be made.

This then, in brief, is the BBM. Kinetic energy is conserved, since all collisions are elastic. Momentum is not conserved, since the mirrors

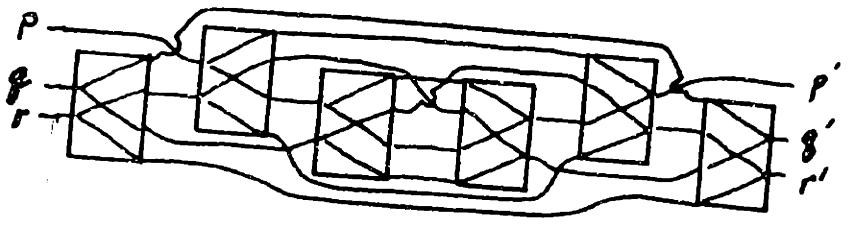


Figure 1.8: Fredkin gate, built out of interaction gates.

are assumed to be fixed (infinitely massive). Since we have shown that Fredkin gates are directly physically implementable (in this idealization), circuits built out of Fredkin gates can now be thought of as the schematics for a BBM circuit.[8]

1.4 Bit-conserving functions

In Section 1.2.4, we gave a proof that reversible logic can be used to perform any computation. This proof was based on the idea that any irreversible combinational circuit can be transcribed into a reversible logic circuit, with extra constant inputs and extra (garbage) outputs. The garbage outputs are then turned into constants by an inverse circuit which undoes everything except for a copy of the answer, which is retained. These reconstituted constants can then be *recirculated* and

[8] There are of course other possible physical realisations of conservative logic gates.

connected as inputs, so that a fresh supply of constants doesn't have to be continually provided from outside of the circuit. All of the non-constant inputs are output along with the answer.

In this section we will show that any invertible, bit-conserving (i.e., sum of inputs conserving) function can be implemented directly in terms of Fredkin gates without producing a copy of the non-constant inputs as part of the output. This result was first demonstrated by D. Silver; the method used here will simultaneously show that only recirculating constants of one kind (all 1's or all 0's) are needed in such circuits.

An immediate corollary of this new result will be that circuits in the billiard-ball model of computation and its various cellular automata analogues (Sections 2.4 and 2.7.1), need have no recirculating constant streams of balls. In this or any similar "physical" implementation of conservative logic wherein 1's are represented by an energetic signal, while 0's are just empty space, the essential role of constants is to provide extra space for the computation—the energy in the inputs to a bit-conserving function is always sufficient to actually perform the computation.

1.4.1 Outline of the Proof

It is clear from Figure 1.1 that if we take any circuit constructed from Fredkin gates and everywhere interchange the connections to each pair q' and r', then the new circuit obtained by this proccess will be logically equivalent to the original circuit, with the role of 1's and 0's in the input and output interchanged. Because of this duality between the roles played by 1's and 0's, we can without loss of generality consider the case where only constants of 0 are to be used.

Our proof will consist of two parts:

I. Any invertible bit-conserving function can be built using a generalization of the conservative-logic *interaction* gate.

II. This generalized interaction gate can always be built from Fredkin gates, making use only of 0-constants.

1.4.2 A generalization of the interaction gate

The interaction gate (see Section 1.3) acts as a *demultiplexer* which conserves the number of 1's in the input: Corresponding to each non-zero input state involving exactly k 1's, there is a set of k output lines. These lines are *all* 1's if the corresponding input state occurs, and are all 0's otherwise.

With two inputs, there are three possible non-zero input combinations, only one of which involves two 1's. Thus we require four output lines (see Figure 1.9). Beside each output line, I've indicated the input state for which that line will be a 1—the output lines have been arranged so that these labels (interpreted as binary numbers) are in numeric order.[9]

[9] This order is different from that of Figure 1.5, but only trivial crossovers are needed to go from that figure to this one.

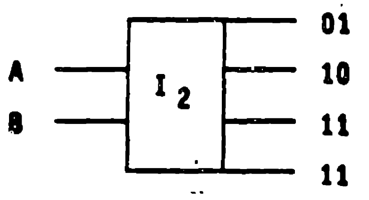


Figure 1.9: Interaction Gate. Each output line is labeled by the input case that makes it non-zero.

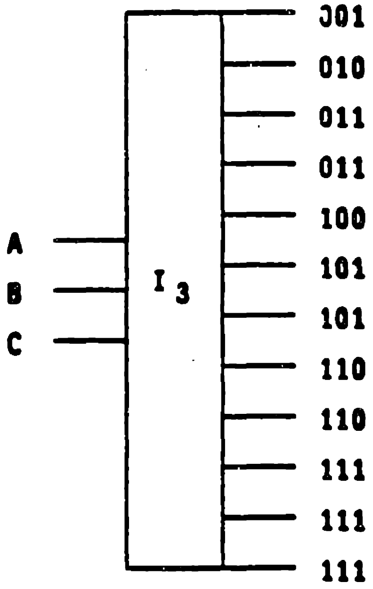


Figure 1.10: A 3 line bit-conserving demultiplexer

With three inputs there are seven possible non-zero input combinations, and we require 12 output lines (see Figure 1.10). If ABC = 101 then both the sixth and seventh output lines will be 1's, all other output lines will be 0's. Similarly for all other input combinations.

In general, we require one output line for each input state containing a single 1, two output lines for each input state containing two 1's, etc.

ABC	XYZ
000	000
001	100
010	001
011	110
100	010
101	011
110	101
111	111

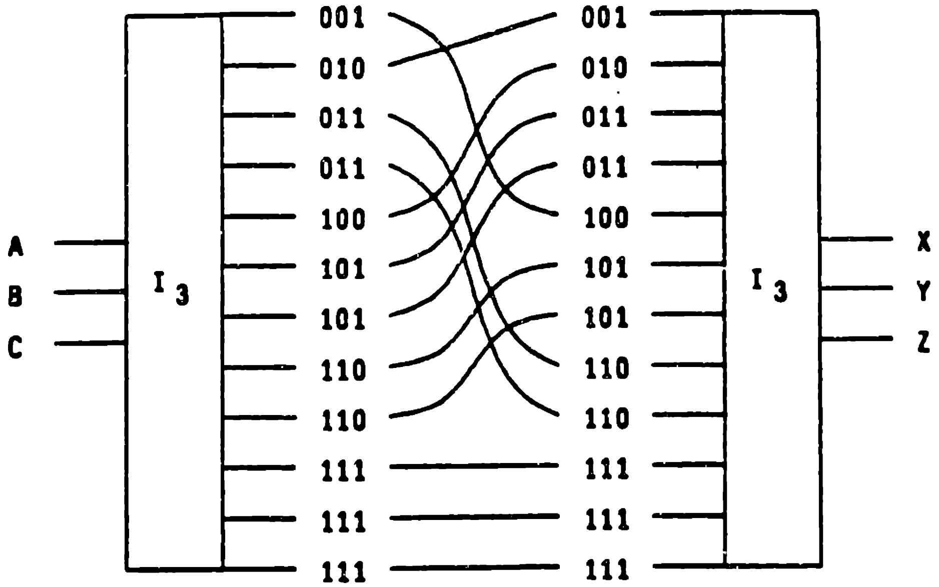


Figure 1.11: SMP gate realized by I_3 gates

Thus for I_n, the interaction gate with n inputs, we require

$$1\binom{n}{1}+2\binom{n}{2}+\cdots+n\binom{n}{n}=n2^{n-1}$$

output lines.

Given an arbitrary bit-conserving invertible function, we can directly implement its truth table using two interaction gates. For example, consider the SMP gate described in Section 1.2.4—its truth table and its implementation in terms of two I_3 gates is given in Figure 1.11. We have assumed in this construction that the mirror image circuit to

I_n is a circuit that performs the logical inverse of I_n, which multiplexes its inputs—in the next section we will note that this is indeed the case when we build the I_n's out of Fredkin gates. Since any set of k output lines corresponding to some input state can be connected to the lines leading to any other state containing k 1's, we can construct any bit-conserving function in this manner.[10]

1.4.3 Building interaction gates out of Fredkin gates

We can consider I_n to be a conservative logic gate having $n2^{n-1}$ inputs and an equal number of outputs: of these *inputs*, all but n are constant 0's, and are not shown in the schematic symbol for I_n.[11]

I_n acts as a 1-conserving de-multiplexer for n inputs. Corresponding to each non-zero input state, there are a set of output lines which are all 1's for that input, and all 0's otherwise.

Given any I_n, we can construct I_{n+1} using Fredkin gates. This is done by considering the cases where the $n+1^{\text{st}}$ input is a 0 separately from those in which it is a 1. Since I_1 is just a wire, establishing this induction shows that all I_n's can be constructed out of Fredkin gates.

[10] Since only wires corresponding to distinct input cases need to cross, no special provisions are needed at crossovers.

[11] We can take advantage of the fact that we are only interested in specifying what happens in the cases where certain input lines are constant 0's to find a simple implementation—this is an advantage of such *constrained* or *underspecified* logic (mathematicians would call this a *partial* function).

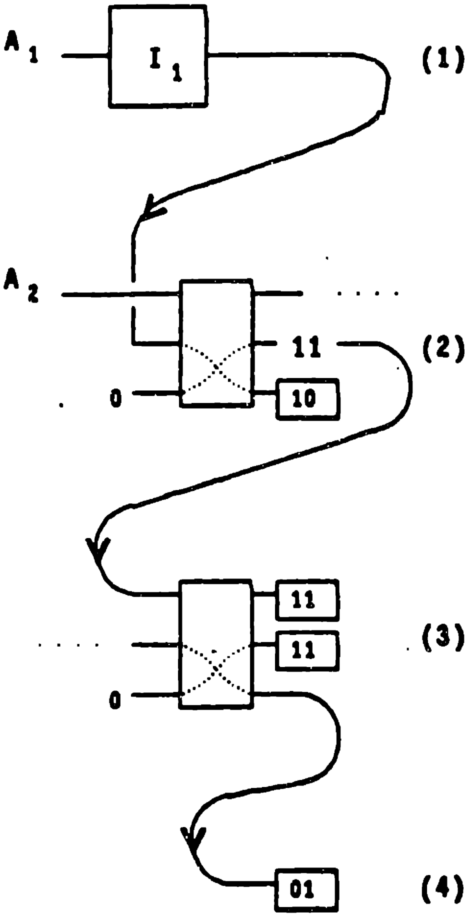


Figure 1.12: Constructing I_n out of I_{n-1} and Fredkin gates, $n = 2$.

We will use the cases $n = 2$ (Figure 1.12) and $n = 3$ (Figure 1.13) as examples, to illustrate the construction.

In each case, the circuit is drawn in four sections, numbered from 1 to 4. The inputs are labeled $A_1, A_2, \ldots, A_n$ and the outputs have $\boxed{\text{boxes}}$ drawn around them.

(1) I_{n-1} demultiplexes all of the inputs A_1 to A_{n-1}. The rest of the circuitry is used to add in the extra input A_n.

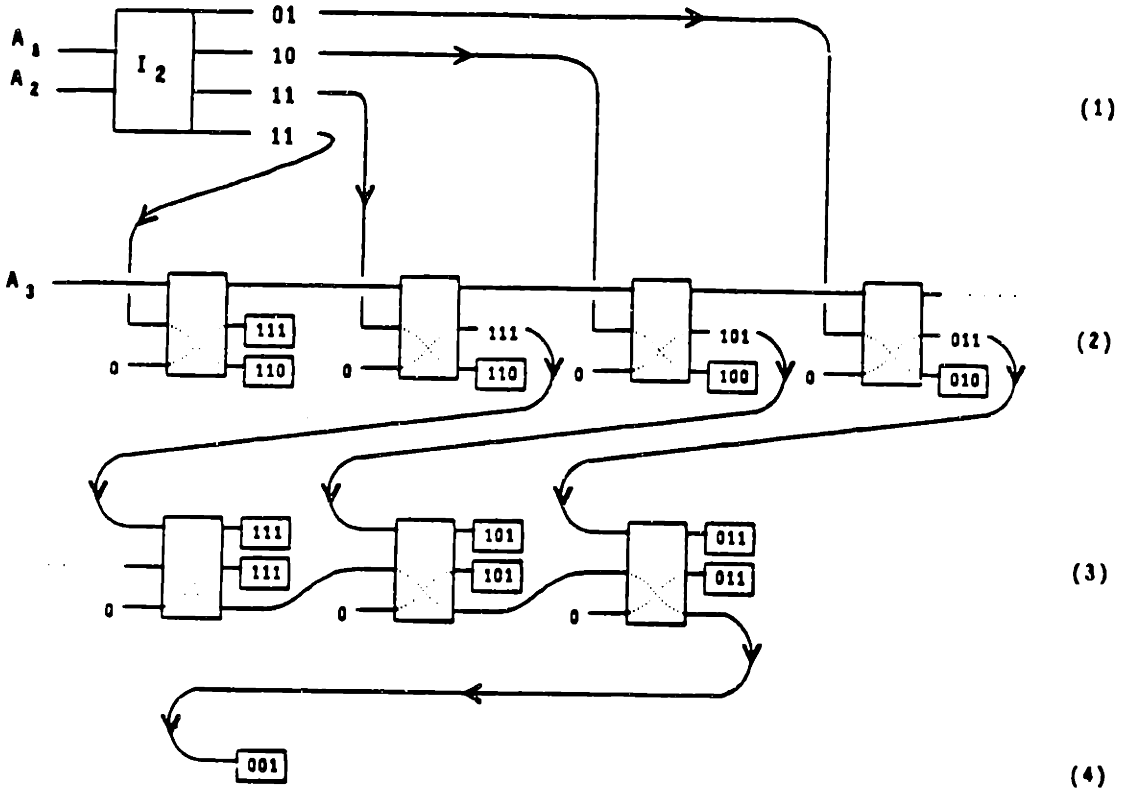


Figure 1.13: Constructing I_n out of I_{n-1} and Fredkin gates, $n = 3$.

(2) Input A_n is used to split each of the outputs of I_{n-1} into two new outputs—the cases where the input is extended with a 0, and the cases where it is extended with a 1. If A_n is 0, then our output should have exactly one output corresponding to each output of I_{n-1}—these are the bottom row of boxed outputs in this section of the circuit. If I_n is a 1, then each output of I_{n-1} that came from an input containing k 1's has been extended to correspond to an input containing $k + 1$ 1's. Thus a representative of each input case must be sent down to section (3) of the circuit to be copied. All other demultiplexed outputs may be output immediately—middle row of outputs in this section of the circuit.

(3) Since no inputs to any of these gates will be non-zero unless A_n is a 1, we can think of A_n as being a 1 in analyzing this section of the circuit. Since the controlling signals for these conditional exchanges correspond to mutually exclusive input conditions, we are able to pass our constant of 1 through all of them and at most one of these will use it up making a copy.

(4) Finally, there is one case that doesn't correspond to any case for which I_{n-1} produced a 1 at some output—this is the case where all inputs to I_{n-1} were 0's. In this case A_n has run through every gate in this circuit and now appears as a 1 at the bottom output of the last gate.

This method allows us to construct I_n from a given I_{n-1} and so, by induction, our assertion is proved.

1.4.4 Corollary

This proof allows us to settle an interesting question about circuits in the billiard-ball model of computation (Section 1.3), and the cellular automata (Sections 2.4 and 2.7.1) modeled after it.

Fredkin's original proof of the universality of the BBM (given in Section 1.3) showed that the BBM could implement any bit-conserving, invertible function. The construction used in this proof required some number of streams of balls (constant 1's) to be supplied as extra inputs to the circuit implementing the function, which were used at some intermediate place in the circuit, and finally regenerated and output as constant 1's along with the results of the computation. Thus these extra 1's could be fed back from the output to the input, and recirculated—no extra 1's needed to be fed into the circuit from the *outside.*

As we saw in Section 1.3, one can construct a Fredkin gate from billiard-ball collisions (interaction gates) without any recirculating 1's. Thus a corollary of our result for Fredkin gates above is that recirculated 1's are never required in the BBM.

1.4.5 Discussion

In the BBM, constants of 0 have a special status, since 0's are represented by empty volume, and so are free of the concerns associated with constants of 1 such as arranging appropriate crossings with other signals and setting up extra mirrors to keep them on some closed path in order to recirculate. Thus it is quite convenient to know that constants of 1 are never required.

The way we have found of avoiding the need for constants of 1 is to use the constants of 0 to provide places for the signals in the input to spread out, until all possible input cases are represented in separate places, at which point the function we wish to implement can be constructed as a lookup table—each input case is wired to the appropriate output case (Figure 1.11). In practice we can usually get by without demultiplexing all cases, but for a random invertible mapping this is what we would have to do[12].

Although we have seen that constants of 1 are not essential in implementing bit conserving functions, they can be very useful. We can often decrease the computation-delay (the time from when the inputs go in to the time the results come out) by using constants of 1, which allow us to make copies of the inputs and of intermediate results, allowing portions of the computation to proceed in parallel.

1.5 Logical heat

When we introduce constraints into our models of computation, it may become possible to make strong global statements that couldn't be made before. If these constraints are artificial, then any newfound ability to make such statements is spurious.

Reversible logic adds the important constraint of reversibility that

[12] Of course all forms of logic have a similar problem with random Boolean functions.

is shared, as far as we know, by the dynamics of all microscopic physical systems. In a reversible model such as the BBM, we also have an additively conserved quantity (the number of balls) which can be identified with the kinetic energy of the system. We will show that in an appropriate context, these two constraints together can play the role of the first two laws of thermodynamics.

The example we discuss is the construction of circuits in the BBM which reclaim some of the energy (balls) tied up in representing unwanted information (garbage). Although we have shown that garbage can eventually be eliminated, and constants restored, by a *mirror circuit* technique, we assume for the sake of this discussion that we have some signals for which this technique cannot be applied (we will discuss at the end some situations in which this would be the case). We will show that there is a maximum efficiency with which any BBM circuit (however constructed) can perform this task, which is closely analogous to the maximum efficiency of an ordinary heat engine. We will then generalize our arguments to show that a similar discussion can be made using any computational model based on reversible logic.

1.5.1 The Impossible Box

Reversibility imposes a strong constraint upon circuits which must deal with random sequences of inputs. Consider for example the schematic BBM circuit in Figure 1.14. This hypothetical circuit has inputs consisting of two random sequences of 1's and 0's. The probability that

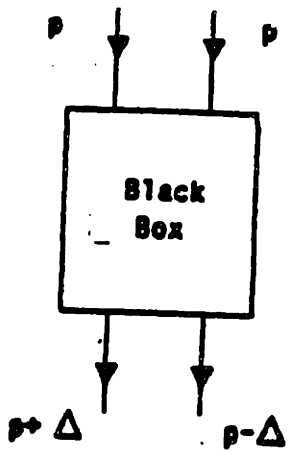


Figure 1.14: A BBM circuit with two inputs where 1's arrive at each timestep with probability p.

any given element of one of these sequences is a 1 is p, the probability of a 0 is $1 - p$. At each time step, the next element in these input sequences enters our circuit.

Given these inputs, our circuit is supposed to produce two output sequences, one of which has (on the average) a fraction $p + \Delta$ of 1's, and the other a fraction $p - \Delta$. In order to accomplish this, we are free to put any BBM circuitry whatsoever inside of the *black box*—it can have feedback and recirculating constants; it can even contain all the circuitry of a general purpose computer. Our question is, can we design a circuit which does what we've described?

The desired circuit would conserve energy: on the average, just as many balls would come out as went in. But the fact that any BBM circuit must be invertible imposes an additional constraint that this circuit fails to meet. In a given number N of time steps there is some number

$$\Omega \sim \binom{N}{pN}^2$$

of distinct possible input sequences that will enter our circuit. With unequal output frequencies, there are *fewer* distinct output sequences possible. Since no invertible function can map a large number of input possibilities into a smaller number of output sequences, no finite BBM circuit can do what the circuit of Figure 1.14 does indefinitely.

As an extreme case, think of the situation with $p = 1/2$ and $p + \Delta = 1$, $p - \Delta = 0$. Here, no matter what sequence comes in, we output constants. None of the information in the inputs is recorded in the outputs, and so we will have trouble if we want to run this supposedly reversible system backwards.[13] If we constructed a reversible circuit that seemed to be doing this, then we could be sure that the missing information must be accumulating inside the *black box*. If we redraw this circuit a bit (Figure 1.15) we see that asking us to construct this circuit is much like asking us to construct a Maxwell Demon: the two loops are like two one-dimensional boxes of gas, initially at equal pressures. The Demon can't create a pressure difference because of invertibility.[14]

[13] To run it backwards we would of course have to reverse the motions not only of the input and output streams, but also of all particles inside of the *black box*.

[14] It was Edward Fredkin who pointed out to me that such a circuit is impossible, and is analogous to a Maxwell Demon.

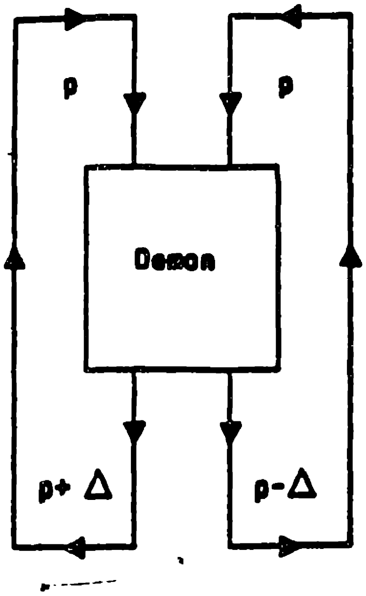


Figure 1.15: A Maxwell demon, creating a 'pressure' difference between two vessels initially at equal pressure

1.5.2 Temperature

We would like to develop an analogy between possible and impossible circuits with probabilistic sequences of input values, and possible and impossible heat engines in thermodynamics. For this purpose, we will define the quantities which will play the roles of entropy and temperature in this analogy.

Given any circuit which has inputs each of which is a random sequence with a probability p_i for a given element to be 1 and $1 - p_i$ for it to be a 0, then the average information-theoretic entropy entering each input during each time-step is[15]

$$\Delta S_i = -p_i \log_2 p_i - (1-p_i)\log_2(1-p_i)$$

If some set of outputs of this circuit are all pseudo-random sequences with probability p_o for a 1 and $1 - p_o$ for a 0, then the average information-theoretic entropy leaving each output during each time-step is

$$\Delta S_o \leq -p_o \log_2 p_o - (1-p_o)\log_2(1-p_o)$$

with equality only if all correlations are neglected. Since we will be considering situations in which as much information as possible is put into each output, we can assume that the equality holds, and also that all outputs are uncorrelated.

For input or output wires which carry a signal which is not a constant, we form the intensive ratio

$$T = \frac{p}{\Delta S} = \frac{-p}{p \log_2 p + (1-p)\log_2(1-p)}$$

of the average number of balls (energy) that pass a given point per unit of time, divided by the entropy carried past that same point by this energy flow in a unit of time. This ratio T is a measure of how efficiently balls which appear in the given wire are being used to represent information. (Notice that this *temperature* is a property of a *wire*, and *not* of an individual ball.)

[15] This is just the limit as $N \to \infty$ of the $\log_2$ of the number of sequences of length N consistent with this probability assignment, divided by N.

1.5.3 Reclaiming Balls

We will consider the problem of designing circuits which take as inputs probabilistic sequences of 1's and 0's, and produce as outputs some number of constant streams of 1's, along with a set of probabilistic output sequences. This is an alternative way of producing some constants of 1 in situations when the mirror circuit technique of Section 1.2.4 isn't applicable. Given our statistical assumptions, we would like to investigate the maximum efficiency with which any BBM circuit can produce such constants.

For the purposes of the analogy we are drawing, we will call a constant stream of 1's *work*—such a stream consists of maximally available energy (the balls have energy, yet they represent no entropy, since their state is always exactly known). Such constants are very useful in the BBM—they allow copies of inputs and intermediate results to be made, thus permitting parallelism to speed up computation. They also simplify logic, and allow it to be more compact, and more similar to conventional irreversible logic.

Suppose for simplicity that we have some number of inputs with a *high* probability p_H of being 1's. We could produce some number of outputs with a *lower* probability p_L of 1's, plus some number of outputs which are always 1. (Our circuit would need to allow for statistical fluctuations, but this is not a problem, as we can easily design circuitry that acts like a ball-reservoir). The situation is illustrated in Figure 1.16.

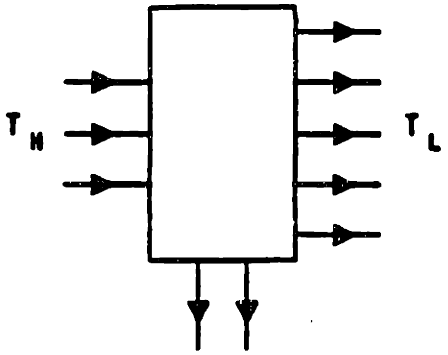


Figure 1.16: Hot (ball-rich) inputs come in from the left, cooler (ball-poorer) outputs leave to the right, while some number of streams of balls (constant 1's) leave from the bottom

It is apparent that we must always have $T_H > T_L$, since the inputs and the outputs both represent the same entropy, but the outputs do it with fewer balls. It is also clear why we can't convert all of the input balls into useful work: we would have no balls left to remember which particular pattern of inputs arrived at T_H.

If we let ΔS_L be the average *total* entropy that comes out at T_L during each time step, and ΔS_H the *total* entropy which enters at T_H during each step, then we must have $\Delta S_L \geq \Delta S_H$ because of the reversibility of the process.[16] Letting ΔQ_H and ΔQ_L be the average *total* numbers of balls entering and leaving in each time step, and recalling

[16] Under our assumptions, ΔS_H exactly characterizes the size of the input ensemble for a long sequence of input values. On the other hand, ΔS_L is a coarse grained entropy which neglects possible correlations: it may overestimate the sise of the output ensemble. If all correlations were taken into account, we would of course always have an equality.

our definition of temperature as the average number of balls per bit in a given input or output wire, this entropy constraint implies that

$$\frac{\Delta Q_L}{T_L} \geq \frac{\Delta Q_H}{T_H} \quad .$$

If we let ΔW be the number of constant outputs produced by our circuit, then on the average we must have

$$\Delta Q_L = \Delta Q_H - \Delta W$$

because of the conservation of billiard balls (or equivalently the conservation of kinetic energy). If we put these two constraints together we have

$$\Delta W = \Delta Q_H(1 - \frac{\Delta Q_L}{\Delta Q_H}) \leq \Delta Q_H(1 - \frac{T_L}{T_H})$$

Under the statistical assumptions given, no BBM "heat engine" circuit can have an efficiency $\Delta W/\Delta Q_H$ that is greater than $(1-T_L/T_H)$. This is the greatest fraction of the balls in the inputs which can be converted into constants.

1.5.4 Generalization

The constraint that the information at the output cannot be less than that at the input of course applies generally to all reversible logic functions. For an unconstrained reversible function, we have also seen that the number of inputs must exactly equal the number of outputs. This additive constraint can play the role that energy did in the discussion above, to give an inequality that applies to *any* unconstrained (i.e., completely specified) reversible function.

We begin by giving a new definition for ΔW. In all reversible logic, constants allow reversible gates to simulate irreversible gates, as we saw in Sections 1.2.3, 1.2.4 and 1.2.5. If no signal reclamation were done, energy would eventually have to be dissipated in order to clean up corrupted "constants" that were used in this manner. Thus the number of constant outputs that come out of any "heat engine" circuit is related to the energy that we are reclaiming, and we will call this quantity ΔW.

It is "number of signalling lines" that is additively conserved, and so if we let ΔQ_{in} be the number of inputs that come into our circuit at every computational step, and ΔQ_{out} the number of information-carrying (i.e., non-constant) outputs, then

$$\Delta Q_{\text{out}} = \Delta Q_{\text{in}} - \Delta W$$

Assuming every input has the same set of probabilities p^k_{in} of being in each possible state k, and similarly for the information carrying outputs and p^k_{out}, we let

$$T_{\text{in}} = \frac{-1}{\sum_k p^k_{\text{in}} \log_2 p^k_{\text{in}}}$$

and similarly for T_{out}. Thus temperature is defined to be the average number of input lines used to represent a bit. Next, we see that the entropy in the input is given by

$$\Delta S_{\text{in}} = \frac{\Delta Q_{\text{in}}}{T_{\text{in}}}$$

(and similarly for ΔS_{out}). Since $\Delta S_{\text{out}} \geq \Delta S_{\text{in}}$, we again have

$$\Delta W = \Delta Q_{\text{in}}(1 - \frac{\Delta Q_{\text{out}}}{\Delta Q_{\text{in}}}) \leq \Delta Q_{\text{in}}(1 - \frac{T_{\text{out}}}{T_{\text{in}}})$$

just as before. With our present definitions of temperature and work, this inequality holds for all unconstrained reversible logic functions. In particular, it will hold for reversible cellular automata.

1.5.5 Discussion

We have investigated the extent to which circuits which are designed to work for inputs which are random sequences can produce outputs which are constants. When might such circuits be useful?

A simple situation where one might want to "concentrate randomness" might arise if some of our reversible circuitry had to perform error correction operations which were relatively rare: this would result in some output of the correction circuitry being essentially (but not exactly) constant. In compressing these error records into fewer signals, we would be limited by the constraints discussed above.

Probably the most interesting situation where our logical heat engines might come into play would be within reversible cellular automata (Chapter 2). These systems can be studied as autonomous digital worlds in which complexity and structure can arise. The usefulness of heat engines to processes running in these cellular automata worlds seems to me to be rather similar to their usefulness in our world.

Chapter 2

Reversible cellular automata

Cellular Automata (CA) are computer-models that embody discrete analogues of the classical-physics notions of space, time, and locality. Their physics-like structure maps very naturally onto physical implementations, making possible extremely efficient hardware realizations (see Chapter 7). This same property of being physics-like makes CA a natural tool for physical modeling[80].

Reversible Cellular Automata (RCA) add the property of microscopic reversibility to the CA paradigm, making possible a still closer correspondence between physical systems and computer models. As an illustration, in Section 2.4 we present an RCA analogue of the classical-mechanical Billiard Ball model of Section 1.3.

The compatibility of computation with CA was first demonstrated by von Neumann[87]. Toffoli[72], in 1977, showed that RCA (of which the only known examples up until then were extremely trivial} could compute, but his models didn't allow Bennett's technique of Section 1.2 to be used to "clean-up" unwanted *garbage* produced during the computation. The RCA analogue of the BBM that I present here is the first CA model which incorporates reversibility in a way which makes computation practicable. All of the results concerning the computing capabilities of reversible logic developed in the previous chapter can be carried over into this model. If this model is implemented in reversible hardware, it can be used to simulate any other 2-dimensional reversible cellular automaton in a local manner.

2.1 Cellular Automata

In CA, *space* is a regular lattice of *cells*, each of which contains one of a small allowed set of integers. Only cells that are close together interact in one *time-step*—the time evolution is given by a rule that looks at the contents of a few neighboring cells, and decides what should change[87]. At each step, this local rule is applied everywhere simultaneously.

The best-known example of such a *digital-world* is Conway's[28] "Game of Life." On a sheet of graph-paper, fill each cell with a 1 or a 0. In each three-by-three neighborhood there is a center cell and eight adjacent cells. The new state of each cell is determined by counting the number of adjacent 1's: if exactly two adjacent cells contain a 1, the center is left unchanged; if three are 1's, the center becomes a 1; in all other cases, the center becomes a 0.

Such a rule gives rise to a set of characteristic patterns that *move* (reappear in a slightly displaced position after some number of steps), patterns that are stable (unchanging with time), patterns that oscillate (pass through some cycle of configurations), and many very complicated interactions and behaviors. The evolution of a given initial configuration is often very hard to anticipate.

One way to show that a given rule can exhibit complicated behavior is to show (as has been done for *Life*[12]) that in the corresponding *world* it is possible to have computers. If you start such an automaton with an appropriate initial state, you will see patterns of digits acting as signals moving about and interacting with each other to perform all of the logical operations of a digital computer. Such a computer-automaton is said to be *universal.*[1] Like other universal computers, a universal cellular automaton can exhibit arbitrarily complex behavior.

We typically show that a CA rule is universal by demonstrating that it supports patterns of cell states that can simulate a universal set of logic elements, signals, and allows the logic elements to be connected together. This implies that we can simulate the cicuitry of any computer, and so the rule is a computer when started from the right initial state. Once the universality of a few CA rules has been established,

[1] Von Neumann[87] was interested in the problem of evolution: Can life emerge from simple rules? He exhibited a CA rule that permitted computers, and in which these computers could reproduce and mutate. In this document, I refer only to the existence of computers when I use the term universal.

other rules may be shown to be universal by demonstrating their ability to simulate one of these universal rules, as we do for example at the end of Section 2.4.

Universal CA rules may be particularly important in connection with fully parallel hardware implementations of CA, since machines based on appropriate universal rules can be usable as general purpose CA simulators (see Chapter 7). If an n-dimensional universal CA rule allows can simulate logic elements which can be connected in n-dimensions, then it can simulate any other CA rule with the same dimensionality in a local manner: a group of cells is used to implement a circuit which simulates one cell of the other automaton, and such groups are interconnected.

In Sections 1.2.1 and 1.2.2 we discussed universal sets of logic gates—if a set of logic gates isn't universal, then no interconnection of such gates can be a computer. In particular, a regular structure built up out of such gates cannot be a computer. Since a CA rule is just some logical function (which can be regarded as a logic gate), only CA for which this function is a universal logic element are candidates for universality. Unfortunately this isn't much of a constraint, since most logic functions are universal (in fact, as we showed in Section 1.2.5, all non-trivial conservative logic gates are universal), but it is occasionally helpful. For example, a rule that can be expressed using only XOR (sum modulo 2) isn't universal (see Section 1.2.2).

2.2 Approaches to reversibility

Any rule that determines the time evolution of finite-state cells and that has a periodic structure in space and in time[2] defines a cellular automaton.

Such rules (also called *transition functions* or *local maps*) are often given by an equation of the form

$$c_{\vec{x},t+1} = f(c_{\{\vec{x}\},t}) \tag{2.1}$$

where we have made use of the following notation: c_t is the complete configuration of cell values at time t, $c_{\{\vec{x}\},t}$ is some portion of this configuration surrounding the cell at position $\vec{x}$ that constitutes the *neighborhood* of $\vec{x}$, and $c_{\vec{x},t}$ is the state of the cell at position $\vec{x}$ at time t. In fact, all CA rules can be put in this form,[3] although this may not be the simplest or most illuminating way to express the rule.

A typical rule of the form (2.1) gives rise to a non-invertible dynamics. For example, the *Life* rule doesn't produce an invertible time evolution: if an area now contains only zeros, did it contain zeros one

[2] That is, the evolution law commutes with a discrete set of translations in space and in time.

[3] If the rules f_1, f_2, up to f_n are used in succession in a cyclic fashion, this periodic time dependence can be removed by using the composition of these n rules as the new rule; a periodic space dependence can be eliminated by regrouping state variables into new cells of the sise of the spatial period; and an n^{th} order time dependence can be hidden by using new cells that contain all the data from n consecutive steps.

step ago, or were there perhaps some isolated ones that just changed? Its impossible to tell.

This typical irreversibility stems from the format of equation (2.1): it describes an evolution that is built out of functions with more inputs than outputs. To make the overall, *global* evolution invertible requires a very careful conspiracy: in constructing the new state from the old, the cells that "see" a given cell as a neighbor must, taken together, retain complete information about its old value. They must take this coordinated action even though each of these cells sees some neighbors that none of the others see.

One way to accomplish this is to use a rule f for which most cells never change, with a large enough neighborhood so that each cell can examine the pattern formed by many nearby cell values. Cells that find themselves in the middle of some particular unchanging pattern can be allowed to cycle through their states without spoiling the reversibility. Such *guarded context* rules, in which the pattern that marked the cells that could change was itself unchanging, were the earliest reversible rules discovered[2].[4]

A much more productive approach is to abandon the format of equation (2.1), and write our CA rules in a form that makes their invertibility manifest, or at least much more readily apparent. There are two techniques known that allow us to do this: the *partitioning* tech-

[4] We can invent non-trivial guarded context rules by using this technique to simulate other kinds of reversible rules we will discuss: cells playing different roles can be suitably marked with unchanging patterns.

niq:ıe, which is based on transition rules that are expressed in terms of reversible logic gates; and the *second order* technique, which is closely related to reversible second-order finite difference schemes. In this chapter we will show that both of these techniques allow us to construct RCA that are universal[5], thus demonstrating that RCA are capable of arbitrarily complex behavior—we will discuss a variety of RCA models of physics in Chapter 4.

2.3 Partitioning cellular automata

Consider a space of cells of some particular size—we'll think of a space consisting of k 2-state cells for definiteness (with periodic boundary conditions, to avoid having to worry about providing a special rule at the boundary).[6] At each step of operation, all k bits are used to construct a new k-bit configuration. The net result of the local CA

[5] A universal RCA is able to simulate *any* computer (given enough time and space) and in a similar manner to irreversible CA, some can simulate any other RCA rule of the same dimensionality in a local manner. Of course no RCA rule can simulate an irreversible CA rule of the same dimensionality in a local manner.

[6] Since we can only build finite systems, we are confronted with the practical problem of deciding what to do at the boundary of the system. If we choose to use a different rule at the boundary, this rule must also be invertible if our overall evolution is to be invertible: the simplest invertible rule to use at the boundary is the identity rule (values at the boundary remain fixed). Usually we completely avoid the problem by using periodic boundary conditions: then there is no boundary. Unless otherwise specified, all of our example systems can be assumed to avoid the problem in this manner.

rule acting simultaneously everywhere is to perform some k-input, k-output function that transforms a given configuration into its successor configuration.

Thus we have an overall function with equal numbers of inputs and outputs—as we noted in Section 1.2.2, such a function can be invertible, provided that it performs a permutation on the set of input configurations. But a cellular automaton is defined in terms of a local rule; how can we ensure that the corresponding global dynamics will be such a permutation? The most straightforward way to guarantee that such a function will be invertible is to construct it as a composition of invertible logic elements. (In fact, all known RCA rules can be written as such compositions).

We refer to CA rules that are based on logic elements with equal numbers of inputs and outputs as *partitioning cellular automata.* The essential feature of these automata is that at each step of the updating, the state variables are partitioned into disjoint groups, and each group is updated as a unit.

As an illustration of the use of the partitioning technique to construct RCA, consider a 2-dimensional space with 2 states per cell. Figure 2.1 shows a Cartesian lattice of cells, divided into 2×2 blocks of cells. We treat each 2×2 block as a conservative-logic gate (see Section 1.2.4), with 4 inputs (its current state) and 4 outputs (its next state). These *gates* are interconnected in an entirely uniform and predictable manner—in applying the rule to the 2×2 blocks, we alternate between using the solid blocking in this diagram for one step, and then

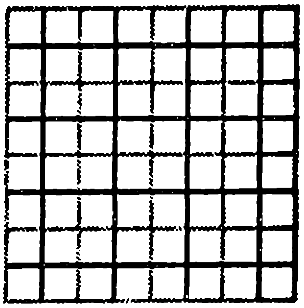
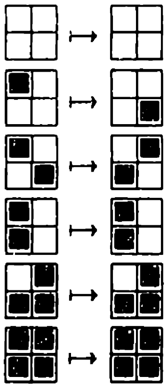


Figure 2.1:

using the dotted blocking for the next.[7]

An example of a conservative rule (one that conserves 1's and 0's) that is reversible is the following:

[7] If f_s is the global rule that applies to the solid blocking, and f_d to the dotted blocking, then $c_{t+1} = f_s(f_d(c_t))$ describes the evolution of a configuration c_t using a time independent rule. By simply regrouping the bits into larger cells, the positional dependence can similarly be removed from the form of the rule, so that it can be written in the form (2.1) if desired.



(2.2)

Here a 0 is shown as an empty cell ( ) and a 1 as a filled-in cell (■). In the case of all 0's or all 1's, there is no choice, they remain unchanged.

A 90°, 180°, or 270° rotation of one of the blocks on the left is mapped onto the corresponding rotation of the result to its right—this rule is rotationally symmetric, and these are all of the possible cases.

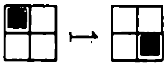
Since each distinct initial state of a block is mapped onto a distinct final state, this rule is reversible. We will find, in the next section, that the automaton corresponding to this rule is universal.

2.4 The BBM cellular automaton

When viewed only at integer time-steps, the BBM (see Section 1.3) consists of a Cartesian lattice of points, each of which has associated with it a value of either 0 or 1, evolving according to a local rule. It would therefore seem to be a straightforward matter to find a CA rule that duplicates this digital time evolution.[8]

Unfortunately, the most direct translation of the BBM into a CA has several problems. First of all, to have separate states of a cell to represent 4 kinds of balls (4 directions) an empty cell and a mirror, and to have the balls absolutely conserved (as they are in the original BBM) would require a standard "change the center cell" rule (equation 2.1) with 6 states per cell, and a 17 cell neighborhood. Such a rule has a very large number of possible configurations for its neighborhood, which makes it unwieldy. Moreover, many of these configurations involve such events as head-on collisions, which were disallowed in the BBM—a CA rule, however, should be defined for all configurations. It is not at all obvious how to extend the BBM rule to these extra cases, and still have it remain reversible.

At the expense of making collisions cause a slight delay, we can get away with the very simple rule (2.2) of Section 2.3 which involves only 2 states per cell in a 4 cell neighborhood, is reversible, and conserves the number of 1's (and 0's) in all cases.

The  (and rotations) case in rule 2.2 is the one that causes an isolated 1 to propagate in a straight line, in one of four directions (depending on which of the four corners of its starting block you put it in). See Figure 2.2. The legend "solid" or "dotted" below each of these automaton configurations tells you whether the grouping of cells into blocks for the next application of the rule is indicated by

[8] A version of the material in this section appears in my paper *Physics-like models of computation*[45].

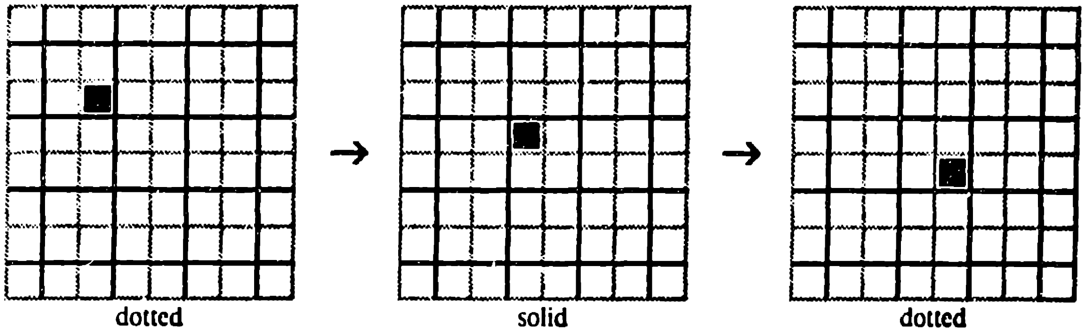


Figure 2.2:

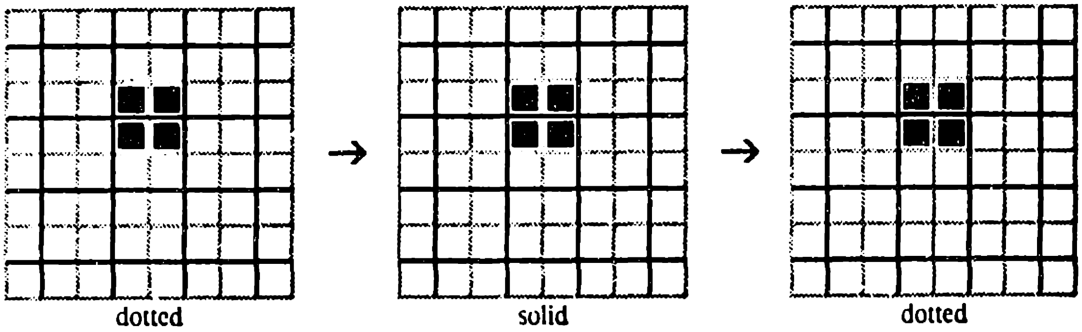
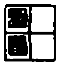
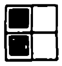


Figure 2.3:

the solid or the dotted lines.

Since  ↦  (and rotations), a square of four ones straddling the boundary of two adjacent blocks will be stable—we will use such squares to construct mirrors, as shown in Figure 2.3. The four 1's straddle two dotted blocks horizontally, then two solid blocks vertically, and then two dotted again.

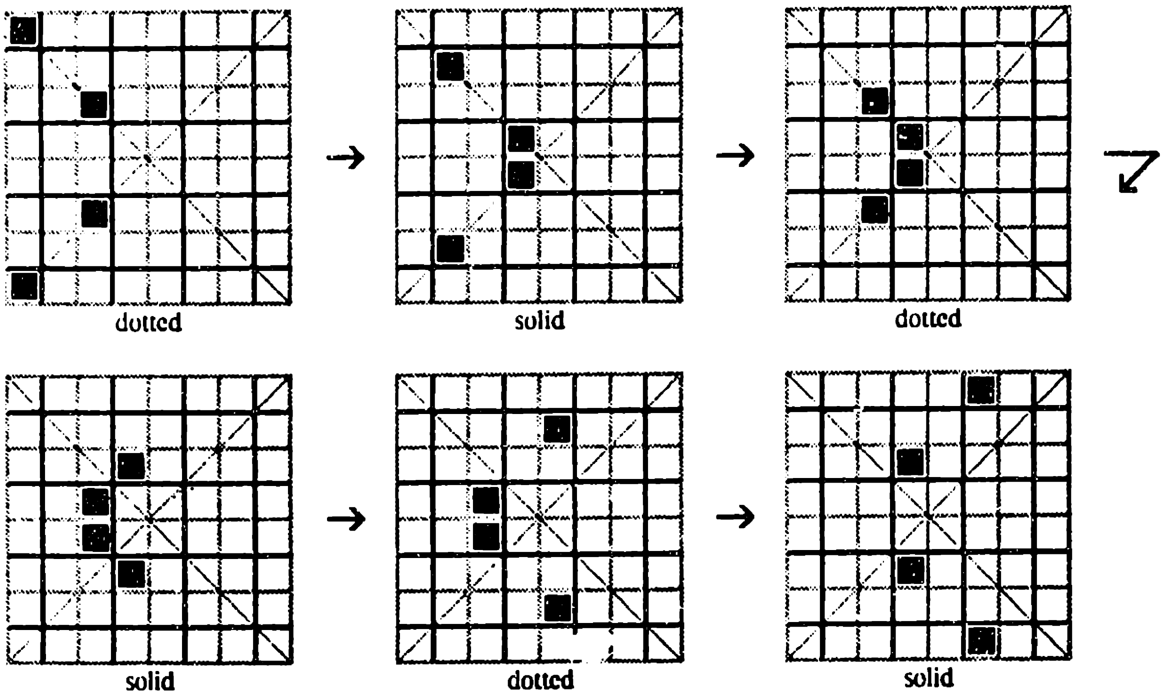
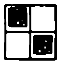
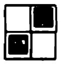
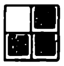
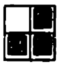


Figure 2.4:

Since  ↦  (and rotations), pairs of travelling ones perform a billiard-ball type collision, as shown in Figures 2.4a through 2.4f. In all of these figures, the paths the 1's were originally following have been lightly drawn in, to show that the AND case shown results in an outward displacement, just as in the BBM. (Unlike the BBM, there is a delay in such a collision, which we'll have to worry about in synchronizing signals).

Finally,  ↦  (and rotations) permits the reflection of dou-

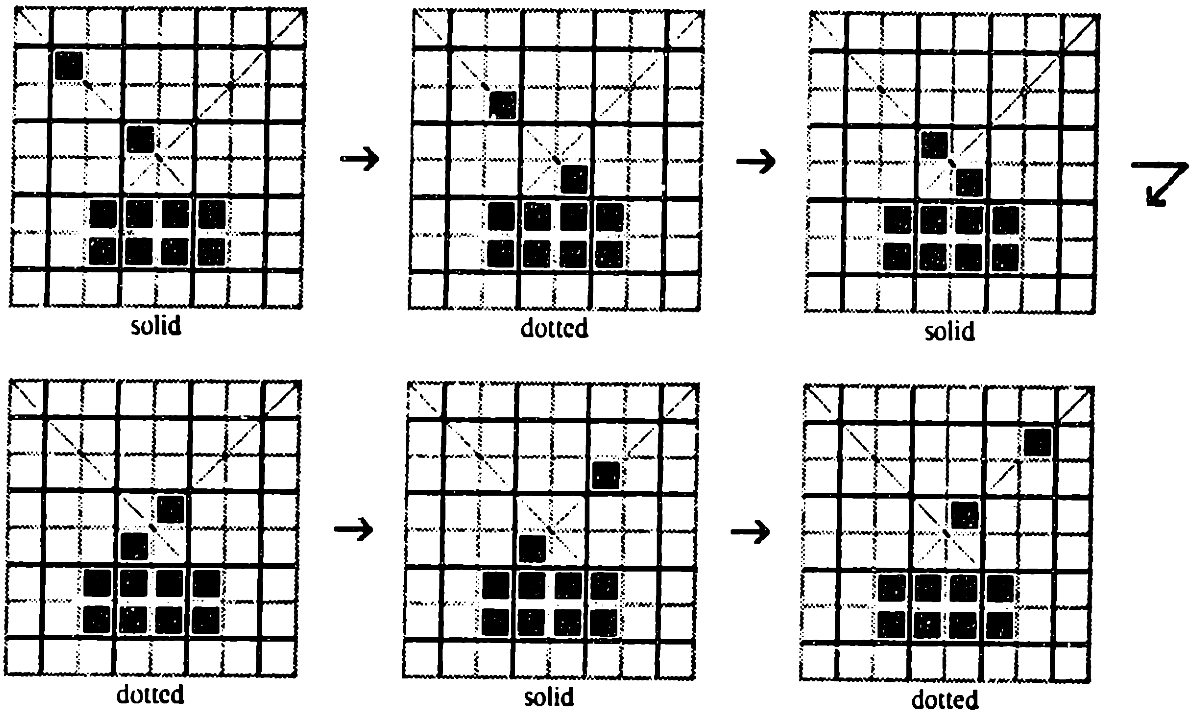


Figure 2.5:

ble signals such as those used in Figure 2.4 by a mirror, as is shown in Figure 2.5. The *mirror* consists of two adjacent stable squares of the sort we introduced in Figure 2.3 (notice that a square is stable no matter what you put next to it—it is *decoupled* from the rest of the evolution). Again, the signal path has been lightly drawn in. After each reflection such as that shown above, the signal has been delayed by a distance of one block along the plane of the mirror (in this picture, the signal winds up one block-column behind where it would have been

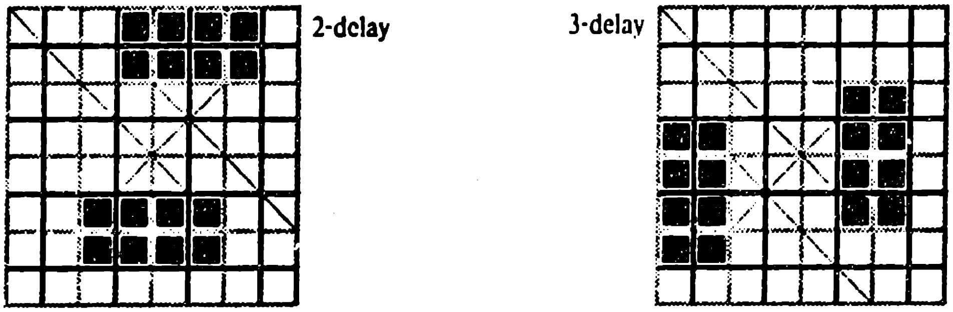


Figure 2.6:

had it not hit the mirror).

In the BBM, such a reflection would cause no horizontal delay. We can compensate for such extra delays, as well as add any desired horizontal delay of 2 or more blocks, by using mirrors to adjust the timing of signals (Figure 2.6). Suppose we want to arrange for two signals to collide, with the plane of the collision being horizontal. If we get the two signals aligned vertically and they are approaching each other as they move forward, they will collide properly. We may have to adjust the time it takes one or both signals to reach a given vertical column by using delays such as those in Figure 2.6.[9]

In order to allow signal-paths to cross without interacting, we use signal timing. By leaving a gap long enough for one signal (2 blocks)

[9] We can tell how many steps a signal will take to traverse a given path (from one position where the signal is moving freely to another) by simply drawing the path joining the two points (including all points that may be visited by at least one 1) and counting how many cells are on the path.

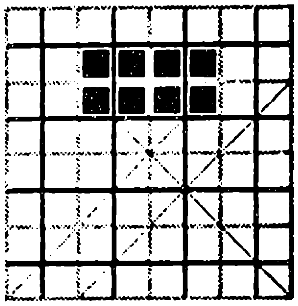


Figure 2.7:

between all signals, we need only delay one of the paths by 2 blocks along the plane of the collision we're avoiding, in order to allow the signals to pass each other harmlessly. This gap is also enough to allow us to separate parallel output paths from a collision (Figure 2.7). After the collision (Figure 2.4) the upper path already has a 1-block horizontal delay relative to the lower path. The mirror introduces a further 1-block delay, and so 'the upper signal passes through the timing-gap left in the lower signal path.

With the addition of some extra synchronization and crossover delays, any BBM circuit can now be translated into a BBMCA circuit. Since the BBM has been shown to be a universal computer, the BBMCA is also.

Figure 2.8 is taken from the screen of CAM-6, the hardware cellular automata machine (CAM) that was designed by Tom Toffoli and myself (see Chapter 7). It shows a BBMCA realization of the circuit

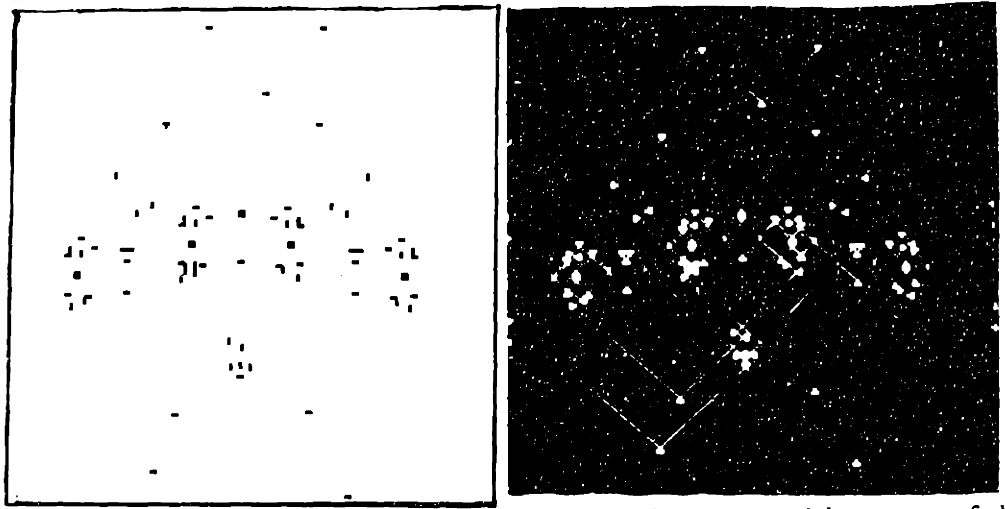
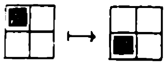
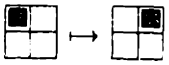


Figure 2.8: BBMCA implementation of a Fredkin gate, with outputs fed back to inputs via signal paths with co-prime lengths, to perform a pseudo-random permutation.

of Figure 1.8 (a Fredkin gate built out of six interaction gates) with outputs fed back to inputs via paths that have co-prime lengths. This circuit generates a very long permutation cycle, and so acts as a pseudo random-number generator. Note that the 1's and 0's of the conservative logic circuit being simulated correspond to *pairs* of 1's (which simulate billiard balls) in the BBMCA, and several steps of the BBMCA evolution correspond to one step of operation of the Fredkin gate being simulated; a pseudo random sequence can be read off by looking at any cell on one of the feedback paths at regular time intervals.

There are many rules similar to the BBMCA that are also universal—for example, if we take the BBMCA rule of fig.2.4 and modify it so that for each case shown, the result (right hand side) is rotated clockwise on the *dotted* steps, and counterclockwise on the *solid* steps (ie.  on dotted steps, and  on solid steps, etc.) then we get another rule that is also computation universal. Its universality can be shown in a direct manner by using this rule to simulate the BBMCA (this rule can simulate a given BBMCA computation isomorphically using eight times as much space, and four times as much time).[10] The possibiliy of such a simulation depends crucially on a scale-invariance property of this and related models, which we will discuss in the next chapter.

2.5 Running backwards

The BBMCA is a reversible system—what is the inverse rule? From table (2.2) it is apparent that if the BBMCA rule is applied consecutively to the same blocking twice, the second application will undo the first, and the net result will be the identity transformation. In general, to undo an entire reversible evolution, we first undo the last step, then the step before that, etc. For the BBMCA, once we have undone a step by reusing the dotted blocking, we have arrived at a configuration that was the result of an updating on the solid blocking. By performing a step on this blocking, we undo another step. Thus to run backwards, we run the evolution exactly as we did to go forwards; only we start by running a step on the opposite blocking to the one that we would use to continue running forwards.

[10] The idea for this BBMCA variation arose out of a discussion with Tommaso Toffoli.

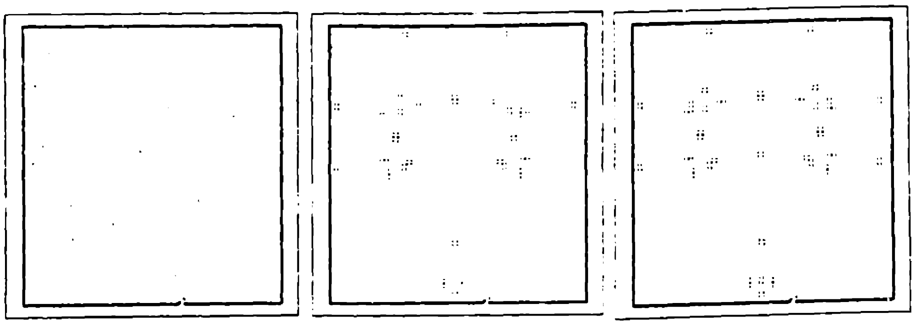


Figure 2.9: Magic-gas experiment: (a) A gas; (b) something happening; and (c) order out of disorder.

What is going on may be more obvious if we consider a single 1, as in Figure 2.2. The direction of travel of the 1 in Figure 2.2a depends upon whether we begin with a step using the dotted blocking, or the solid blocking: it will travel in one of two opposite directions. By starting to run with the wrong blocking, we *reverse the motion of all "particles,"* and the system runs backwards.

In Figure 2.9, we show three stages in the evolution of a BBMCA "gas" in a box constructed of mirror-blocks. Initially (Figure 2.9a) we have a random-seeming gas of particles. For a truly random distribution of 1's and 0's we would expect a very dull evolution, since it is a maximum entropy state, and this is a reversible rule (see Section 2.8.1). As we run this experiment on CAM-6, we see the system begin to simplify (Figure 2.9b), and finally turn into a circuit with a "ball" bouncing around outside the box. What we have of course done is started with a fragile circuit inside the box (see Section 3.1) which uses dynamic mirrors which can be destroyed by a mis-timed collision (Figure 2.9c) and introduced a particle through a small hole in the box. This particle randomized the circuit, turning it into a gas of particles; we stopped the system, reversed all the velocities, and saved the configuration—this is the configuration of Figure 2.9a. This makes an amusing demonstration of the exactness with which we can reverse the motion of particles in an RCA evolution to produce atypical random-looking gases; in Chapter 4, we will discuss more serious uses of RCA gases based on rules closely related to the BBMCA rule.

2.6 Relationship of BBMCA to Conservative Logic

The interaction gate of Figure 1.5 (a schematic representation of a BBM collision) has two inputs and four outputs. If we wish to consider it to be a conservative-logic gate (one that conserves both 0's and 1's) then we must regard it as a gate with four inputs and four outputs, two of the inputs being constrained to always be 0's.

The gate upon which the BBMCA is based also has four inputs and four outputs. Is there some connection here? Let us redraw the BBMCA rule in a different form:

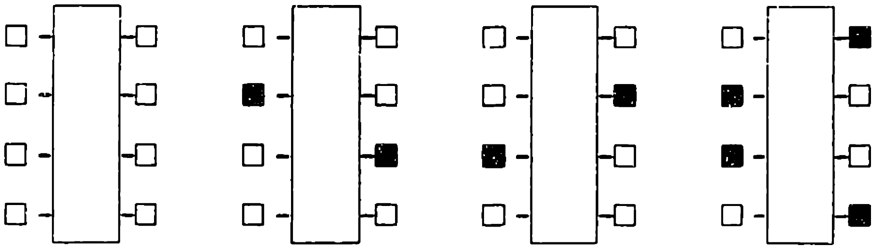
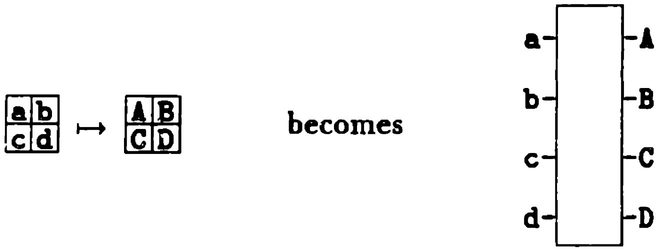


Figure 2.10: Remapping of a BBMCA block.



Here the mapping of input variables onto output variables of the BBMCA rule has been redrawn as if the inputs all arrive and leave in a vertical column. If we use this correspondence to draw the four possible cases with $a = d = 0$, drawing □ for 0, ■ for 1, and showing each input/output case, we get the mapping of Figure 2.10, which is logically the same as the interaction gate. Thus rule (2.2), the BBMCA rule, can be regarded as a completion of the definition of the interaction gate for cases that don't correspond to the constraints of a BBM collision.[11]

[11] Recall that we completed the interaction gate definition in a different way in Figure 1.9, and we couldn't complete it at all in terms of the behavior of billiard balls. This suggests an advantage of using incompletely specified (constrained) logic elements: circuits designed in terms of such elements will work regardless of how (or even if) the unused cases are defined. This gives a lot of freedom to whoever has to find a physical (or even a logical) implementation.

2.7 Second-order cellular automata

In the preceding sections we have discussed CA that are based on logic gates with equal numbers of inputs and outputs. Properties such as reversibility and conservation of 1's and 0's that were given to the gates were inherited by the global evolution; however, such partitioning schemes can equally well give rise to an irreversible evolution if we base them on an irreversible gate (we give an interesting irreversible example in Section 4.4).

It turns out to be very easy to find a class of CA laws that are *always* invertible, simply by virtue of the form of their defining equation.

Consider first the following finite difference equation, with u_t a real variable:

$$u_{t+1} = f(u_t) - u_{t-1} \tag{2.3}$$

If you want to compute u_{t+1}, you must know u_t and u_{t-1}—these two constitute the complete *state* of the system. For what functions f will the time evolution be invertible?

$$u_{t-1} = f(u_t) - u_{t+1} \tag{2.4}$$

Therefore any f at all will do![12] Knowing u for two consecutive times allows you to calculate any preceding or any succeeding value of u (To my knowledge Fredkin[24] was the first to study reversibility in finite-difference equations of this sort).

[12] Assuming integer addition and subtraction is done without error, if such an equation is iterated on a digital computer, its time evolution remains *exactly* reversible, despite roundoff and truncation errors in computing f.

The generalization to CA is straightforward—let u in (2.3) be replaced by $c_{\vec{x}}$, the contents of the cell at position $\vec{x}$ in our automaton:

$$c_{\vec{x},t+1} = f(c_{\{\vec{x}\},t}) - c_{\vec{x},t-1} \tag{2.5}$$

where $f(c_{\{\vec{x}\},t})$ is any function involving the contents of cells near position $\vec{x}$, at time t, and the difference is taken modulo the number of allowed cell values.[13] If we let the state of a cell correspond to its contents in two successive steps, then (2.5) can be reexpressed in the form (2.1), but its reversibility is not manifest.

To give an example using a two dimensional Cartesian lattice with one bit of state in each cell, let the neighborhood $c_{\{\vec{x}\},t}$ consist of the cell at position $\vec{x}$ and its four nearest neighbors:

$$f(c_{\{\vec{x}\},t}) = \begin{cases} 0 & \text{if all 5 neighbors are zeros,} \\ 1 & \text{otherwise} \end{cases} \tag{2.6}$$

Figure 2.11 shows the state of a 256×256 periodic space after several thousand steps of dynamical evolution: it started from a configuration that was all 0's except for a 16×16 region in the center that was all 1's in both the past and the present. The block is still visible, because of a conservation property of this rule (such conservations are discussed in Section 3.2.1). It was curiosity about the long-time behavior of

[13] Differences mod-k and logical functions can always be re-expressed as ordinary polynomial functions. For example, if A and B are binary variables, then $(A-B)^2$ is the same as $A+B$ (mod2), $1-A$ is the same as NOT(A), $A \times B$ is the same as AND(A, B), etc. Thus (2.5) is equivalent to an ordinary real-variable finite difference equation with integer initial conditions.

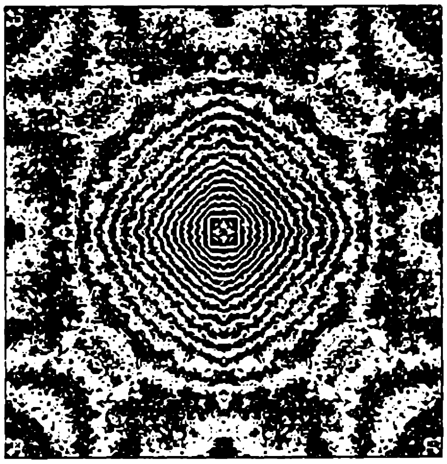


Figure 2.11: State after several thousand steps of a reversible second-order evolution that started from a block of 1's in the center.

this particular rule that started Tommaso Toffoli and I on the road to building cellular automata machines.

How do we run our second-order systems backwards? Since the form of the inverse equation to (2.5) is the same as that of the direct equation:

$$c_{\bar{x},t-1} = f(c_{\{\bar{x}\},t}) - c_{\bar{x},t+1} \tag{2.7}$$

the evolution governed by such an equation can be inverted by simply exchanging the information corresponding to the two configurations that make up the state, and continuing to use equation (2.5) to govern the evolution. If you think of the two consecutive configurations that make up the state of the system as being like two consecutive snapshots of some physical system, then it is quite intuitively satisfying that we make the system run backwards by exchanging the snapshot corresponding to the past with the one corresponding to the present:

this time reversal operation is quite analogous to ıeversal of all particle momenta.

Many second order reversible rules of the form (2.5) can be recast in a sort of Hamiltonian form. A conserved "energy" function is derived, and the evolution rule becomes: make all changes in the configuration that leave the energy unchanged. This is discussed in Section 3.3.

Second-order reversible rules can also be constructed using operations other than subtraction in an equation such as (2.5). You can even let the decision of which operation to use depend on the neighbors at time t. In the most general second-order reversible rule, the neighborhood at time t is used to select a permutation on the set of allowed cell values. The cell applies this permutation to its state at time $t-1$ to construct its next state.

2.7.1 Second-order, reversible, universal automata

Here we will give two examples to demonstrate the ability of our second-order scheme to support universal computation. I constructed the RCA model that will be described first long before the BBMCA, but it is much less elegant. It is also based on the interaction gate (Figure 1.5), but this time the interaction is attractive (this is logically still represented by the same gate). This rule will be incompletely specified: we will only specify the cases that are needed to demonstrate its universality.

As a second example, we will present a second-order rule that can very directly simulate the BBMCA, and so demonstrates the ability of second-order RCA to be universal in a rather simple fashion.

A 3-state rule

For our first example, we begin with a 2-dimensional Cartesian lattice, this time with 3 states per cell, which we can designate as -1, 0, $+1$, and which we will draw as '\', blank, and '/' respectively in diagrams.

The time evolution will be given by equation (2.5), with the neighborhood $c_{\{\bar{x}\},t}$ chosen to be the nine cells in the 3×3 region of the configuration centered on $c_{\bar{x},t}$, and '$-$' is taken modulo 3.

For each possible configuration of the neighborhood, f will return a value of -1, 0, or $+1$. Just as head-on collisions never arise in BBM computations, many configurations of this RCA need not arise in order to *build* a universal computer. We will leave these cases undefined—each choice for these cases defines a distinct universal RCA.

An isolated '/' or '\' will correspond to a travelling billiard ball—if only the cases defined here arise, the number of such *balls* will be conserved. An isolated '/' will propagate along a positively sloped diagonal—its evolution will be governed by the following cases:

```
000  000  /00  000  0/0  000  000  000
000  0/0  000  000  000  /00  00/  000
000  000  000  00/  000  000  000  0/0
```

all return a 0 as the value for f;

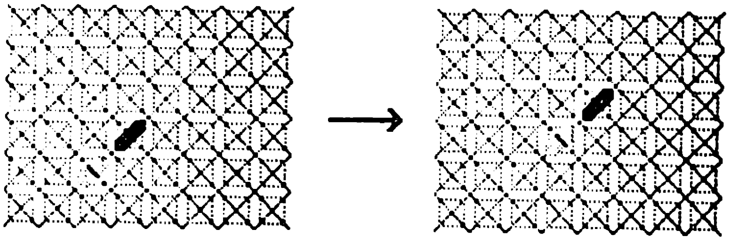


Figure 2.12:

```
00/   000
000   000
000   /00
```

both yield a value of '/' (ie. +1). A sample time evolution (using halftones to show a cell's contents at time $t-1$ and solid lines for time t, with diagonals lightly drawn through all cells) is given in Figure 2.12. Intuitively, this rule at time t tries to make the '/' travel both forwards and backwards along its diagonal—subtracting away a '/' where it was at time $t-1$ just leaves a '/' in the forwards direction.

We will define this rule to be rotationally symmetric. It will be helpful to adopt the following convention: the 90° clockwise rotation of

```
000                \00
000                000
/00   →   /   is   000   →   \
```

Inversions are defined analogously. Thus an isolated '\' will follow a negatively sloped diagonal path if the propagation of signals is governed by the cases:

```
000  000  /00  0/0              000
000  0/0  000  000              000
000  000  000  000   →  0,      /00   →  /
```

(and rotations and inversions). For compactness in writing the complete rule, we adopt the convention that inversions as well as rotations of the cases given are mapped onto the corresponding inversions or rotations of the result given.

These cases become zero:

```
\\\  \\0  \\0  \\0  \\0  \\/  \\/  \0\  \0\  \0\
000  000  /00  /00  //0  000  /0/  000  /00  /00
///  //0  000  00/  00/  //\  /\\  /0/  000  00/

\00  \00  \00  \00  \00  \00  \00  \00  \00  \00
000  /\0  /\/  /0\  /0\  /00  /00  /00  /00  /00
/00  000  000  000  00/  \00  \0/  000  00/  0/0

\00  \00  \00  \00  \00  \00  \0/  \0/  \0/  0\\
/00  /0/  /0/  //\  //0  //0  000  /0\  /0\  000
0//  000  00/  000  000  00/  /0\  \0/  000  000

0\\  0\0  0\0  0\0  0\0  00\  00\  00\  00\  00\
000  000  000  000  /0/  0\0  00\  000  000  00/
0//  000  00/  0/0  0\0  000  00/  000  00/  000

000  000  000  /\\  /\\  /0\
0\0  000  /0\  000  /0/  000
000  000  \0/  \//  \\/  \0/
```

These cases become one:

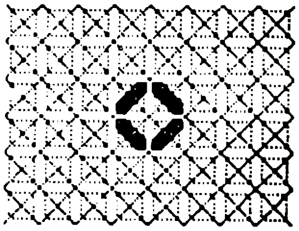


Figure 2.13:

```
\00   \00   \00   \00   \0/   \0/   \0/   \/\   0\0   0\/
/\0   /00   /00   /00   /0\   /00   /00   /00   /\0   \00
\/0   /\0   /00   /0/   00/   000   00/   000   \/0   /00

0\/   0\/   00\   000   000   000   000   000   000
00\   000   /\0   000   000   /\\   /\0   /\0   /\/
000   /00   \/0   /00   /0/   \//   \/0   \//   \/0
```

(plus rotations and inversions). There are 2617 undefined cases.

Using this rule, a mirror looks like the configuration given in Figure 2.13.[14] We needed to define certain cases just to allow a mirror to remain unchanged when no signals are nearby. A signal bouncing on a mirror is shown in Figure 2.14. (Notice that there is no horizontal delay, as there was in the BBMCA). If this signal had been shifted one column to the right, it would have passed the mirror unaffected.

In Figure 2.15 we have put some mirrors near a place where signals might collide, so that (with its small neighborhood) this rule can

[14] These figures were taken from the screen of a Lisp Machine, which was used to simulate this rule to verify that it works.

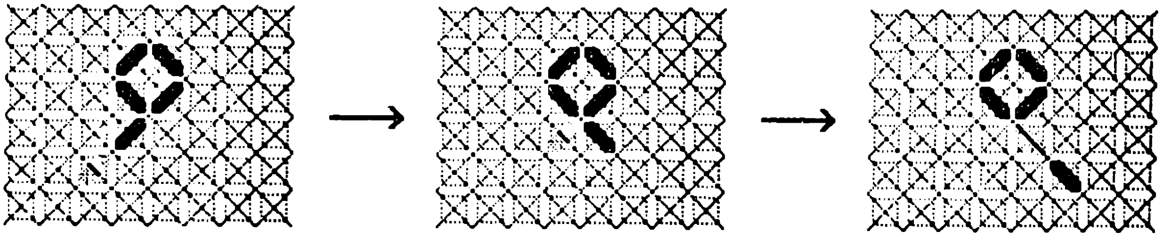


Figure 2.14:

simulate an attractive collision—the signal paths will be displaced inward in a collision, rather than outward as in the BBM. (If a signal arrives on just one path, it goes through without any displacement). Two such gates, back to back, can be used to make signals cross over without affecting each other (Figure 2.16a and b). Since all collisions occur without any delay along the plane of the collision, considerations of synchronization are very similar to those in the BBM. The proof of this automaton's universality is essentially the same as for the BBM.

An embedding of the BBMCA

To give a simpler derivation of a universal second-order RCA, we can begin with the BBMCA rule. If f_s is the global rule that applies to the solid blocking and changes an entire configuration into the next configuration, and similarly f_d applies to the dotted blocking, then we can describe the BBMCA evolution by

$$c_{t+1} + c_{t-1} = f(c_t) \tag{2.8}$$

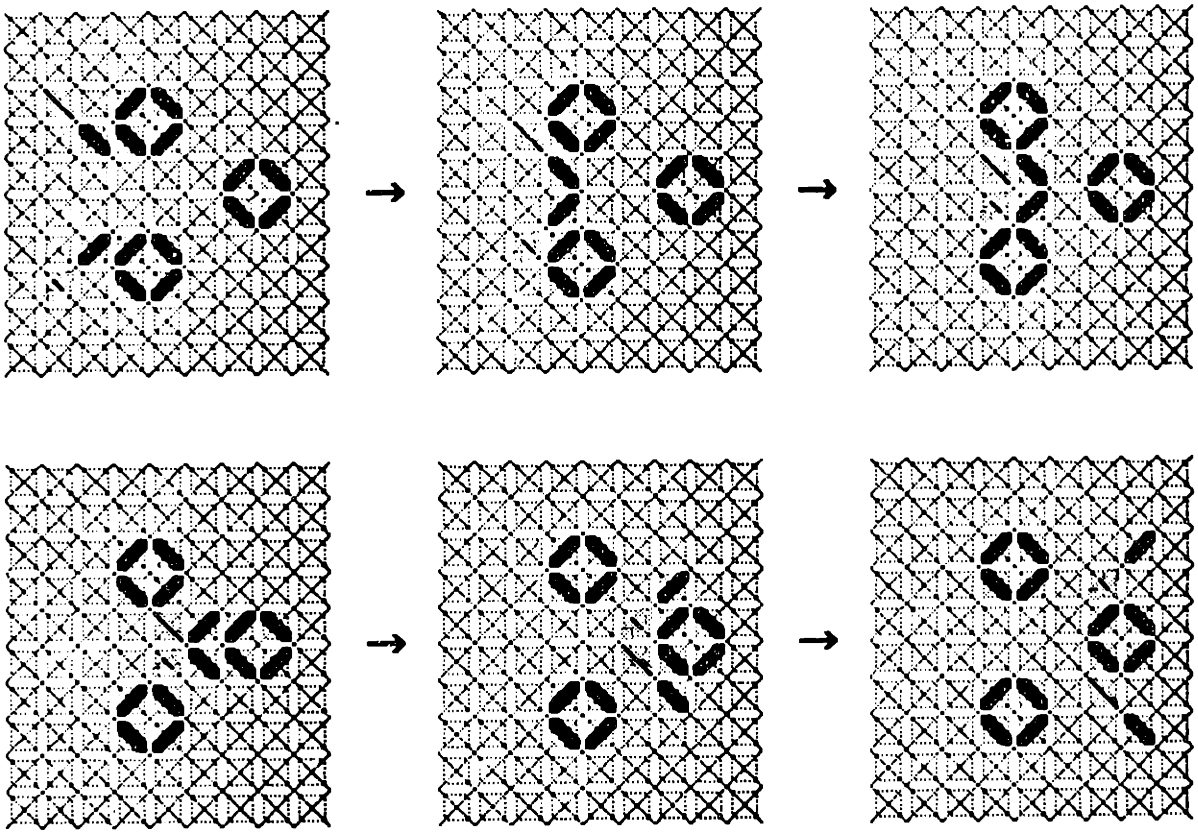
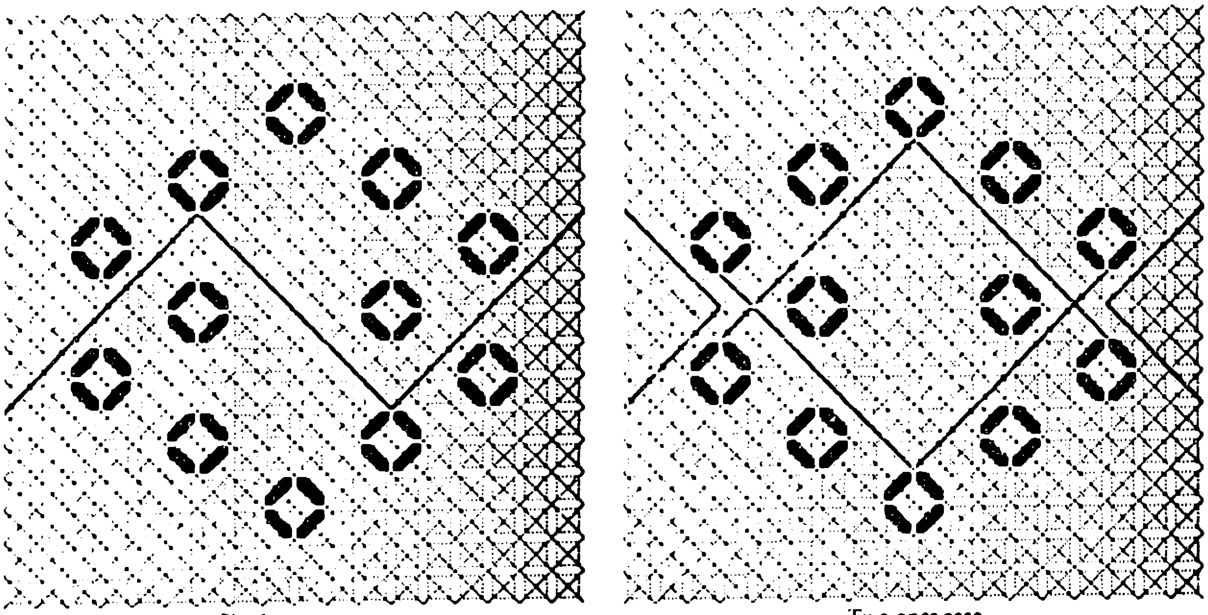


Figure 2.15:



Single one case

Two ones case

Figure 2.16:

where $c_{t+1}+c_{t-1}$ is taken to be the configuration obtained by performing the cell-by-cell sum (modulo 2) of the configurations c_{t+1} and c_{t-1}, and $f(c_t) = f_s(c_t) + f_d(c_t)$ is also such a sum.

In other words, if we add the forward evolution to the backwards evolution, we get a second order evolution which is no longer time dependent, since we've used both partitions. Each cell is at the intersection of two blocks—one from the dotted blocking, and one from the solid blocking. By using all seven of the values in the two blocks, we can determine what the new value of the cell at the intersection would be if it was updated as part of either block, and thence the sum modulo 2 of these two values which is the value that should be returned by f. Thus equation (2.8) can be rewritten in the form (2.5) with a 3×3 neighborhood and a dependence on the parity of the center cell's position that is needed for the rule to know which seven of the nine cells in the neighborhood to look at. This final rather trivial spatial dependence can be eliminated, if we want to use this as a proof of the universal computing ability of rules of the form (2.5), by adding one bit to the state of every cell, and starting the system out with the values of these added bits reflecting the parity of each cell's position.

We can use this new rule to run any configuration exactly as the BBMCA would: we specify one configuration arbitrarily, and then run one step of the BBMCA evolution on this configuration to get the second configuration needed by our second-order rule. Our second-order evolution will now generate *exactly* the same sequence of configurations that the BBMCA system would. We mustn't forget, however, that

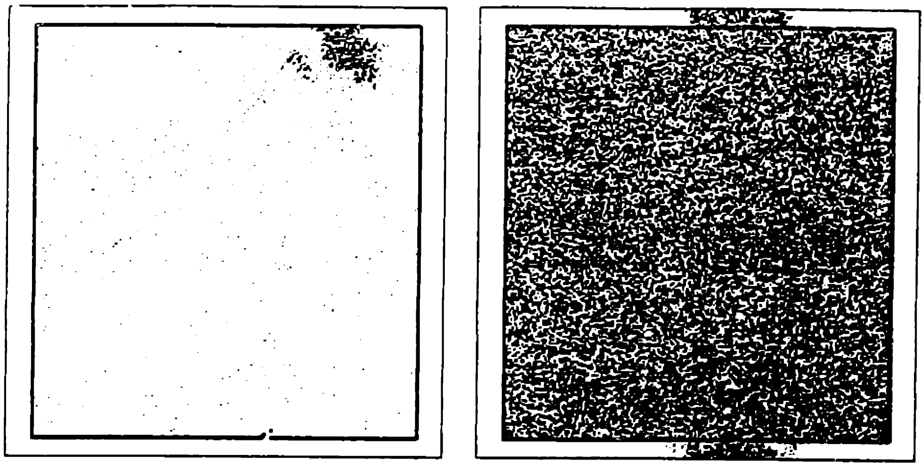


Figure 2.17: Second-order simulation of BBMCA, with an anomaly.

we are only using a carefully constrained subset of the possible initial states of our second-order system. Figure 2.17 shows two frames from the evolution of such a system which was simulating a BBMCA evolution (a gas which, for the BBMCA, would have been in a maximum entropy state) when we changed a single bit in one of the configurations. This resulted in there being a place where there was a particle in the present, but no particle it could have come from in the past. The first frame shows the situation shortly after the bit was changed; the second frame shows the situation a few hundred steps later. We call this simulation, "The end of the world."

2.8 Consequences of reversibility

Reversibility is a very deep and subtle property for a dynamical system to have—it has many consequences. In Section 1.5, we discussed the way that reversibility imposes a constraint which is analogous to the second law of themodynamics. In Section 2.8.2 we will discuss the striking coincidences that must occur when an RCA is reversible. In Chapter 3 we will discuss conservation laws in RCA, which in some sense are an expected consequence of reversibility (after all, RCA must always retain enough information to reconstruct their initial state). In Section 5.1, we will see a particularly striking consequence of reversibility: the macroscopic arrow of time that is relevant to the evolution of processes within a finite RCA may initially agree with the order in which the updating produces new configurations, and then later point in the *opposite* direction! This is closely related to the discussion of the next section.

2.8.1 Entropy in RCA

If we fill the cells of our automaton with randomly chosen binary values and then evolve it according to the *Life* rule, we see a complex ebb and flow of structures and activity, with so-called *gliders* arising here and there, moving across clumps of zeros, and then being drawn back into a complex boiling *soup* of activity, or perhaps rekindling complicated interactions in an area which had settled down into uncoupled, short period oscillating structures.

If, instead of the Life rule, we follow some invertible time evolution, we invariably find that, at each step, the state of the automaton looks just as random as when we started.[15]

[15] Spatial correlations will not arise if they are initially absent, but time corre-

This is expected from a simple counting argument: Any given number of steps of evolution of an RCA rule performs a permuιation on the set of configuration states—each distinct initial state is mapped onto a distinct final state. Since most of the possible states look "random," a typical random-looking state must be mapped by this evolution onto another random-looking state—there just aren't enough simple-looking states to go around.

This is not meant to imply that RCA are less interesting than irreversible CA. Starting an RCA from a random state is like starting a thermodynamic system in a maximum entropy state—its not allowed to get any simpler since its randomness can't decrease, and it can't get more complicated, since its already as random as it can be, and so nothing much happens.

If we start an RCA from a very non-random state (eg. some small pattern on a background of zeros, as we did in Figure 2.11) then we can have an interesting time evolution. If we choose a rule and an initial state that allow information to propagate, then what tends to happen is that the state of the RCA becomes more and more complicated. More precisely, if each state of the automaton is viewed as a "message," with the contents of the cells being the characters of the message, and if only local measures of correlation are applied, then the amount of information[16] in successive messages tends to increase. For example, if

lations are often very evident, and are characteristic of the particular rule being employed—conservations are often particularly apparent (see Section 3.2.1).

[16] For a discussion of the information content of a message, cf.[67]

we let n_i be the fraction of all cells that are in state i, then the quantity

$$s = \sum_i -n_i \log_2(n_i) \tag{2.9}$$

is the average information content of a cell,[17] using the most local measure of correlation (none), and for almost any RCA rule and almost any initial condition is found empirically to eventually increase to a maximum value which persists indefinitely.[18] This suggests that most RCA rules, within the constraints of the invariant quantities that they preserve (see Chapter 3), perform a sufficiently complicated and nonlinear transformation on neighborhoods that a coarse-grained entropy such as this[19] tends towards a maximum *equilibrium* value, at least for systems which don't cycle first.

Of course the automaton is really only repeatedly encrypting its state, and so if all correlations are taken into account the amount of information really never changes. What happens is that the automaton will introduce some redundancy into the message, and use more cells to

[17] The limit as $N \rightarrow \infty$ of the $\log_2$ of the number of configurations in an N-cell space that are compatible with this set of n_i's, divided by N.

[18] A less local measure would be, for example, to let the p_i's in equation (2.9) be the frequencies of all possible 2×2 blocks (e.g., there are 16 different kinds for binary-valued cells). This would certainly be a more interesting quantity than the most local measure for rules based on conservative logic (i.e., binary valued cells, and rules that conserve the total numbers of 1's and 0's).

[19] A coarse-grained entropy that is more like that of classical statistical mechanics would involve smearing some of the fine details of configurations. For example, we could divide our system into $k\times k$ blocks, and lump together microstates that can be transformed into one another by permutations of sites within these blocks.

encode the same information. Information that was initially localized becomes spread out as correlations between the states of many cells, and it becomes very difficult for a locally invertible evolution to put the redundant pieces back together.[20] To use an analogy, an invertible mapping could change two copies of this chapter into one copy, and several sheets of blank paper. Two separate invertible mappings, each acting only on one of the copies, could not accomplish this end.

A less direct argument that points to increasing complexity is just a variant on our earlier counting argument: For an RCA to complete a cycle, it must "find" its initial state—it can't repeat any other state before repeating that state, since each state has a unique predecessor. Since there is generally nothing driving an RCA towards its initial state, RCA tend to have very long dynamical orbits (think of the recurrence time for a BBMCA "gas" of particles, started with most particles in a clump). But since there are relatively few states that have a simple structure, a long orbit implies that the system must eventually make use of more complicated states.

We have already discussed in Section 1.5.4 a rather general class of reversible logical systems in which an interaction characterized by a probability distribution can be analyzed in thermodynamical terms. The fact that isolated subsystems in an RCA tend towards maximally-disordered states which then persist for a very long time should make it

[20] Equation (2.5) generates a locally invertible time evolution. If we know the values of cells near position $\vec{x}$ at two successive times, we can tell what the preceding value of the center cell was.

possible to use similar thermodynamic reasoning to put constraints on what processes operating within an RCA between two such equilibrium subsystems can do.

2.8.2 The charming circle

In this section, I will present a consequence of reversibility which is present in RCA, but is more striking when presented in terms of a reversible finite-difference scheme. The model I will use has been studied by Fredkin, and the invariant associated with it was discovered by Feynman. This dynamical system was actually first discovered by Marvin Minsky by accident, when he made a mistake in a program to draw a circle (we will not follow his original derivation).

A simple equation to generate points on a circle would be

$$z_t = x_t + iy_t = z_0 e^{i\omega t}$$

For $t = 0, 1, 2$, etc., this would generate points (x_t, y_t) at angular separations ω around a circle of radius $|z_0|$. Points can of course be generated by multiplication of earlier points:

$$z_{t+1} = e^{i\omega} z_t \tag{2.10}$$

or (going backwards)

$$z_{t-1} = e^{-i\omega} z_t$$

Taking the difference of these two equations gives us a second order equation

$$z_{t+1} - z_{t-1} = 2i \sin(\omega)\, z_t \tag{2.11}$$

We can transform this into an equation that has only real coefficients by letting $s_t = i^t z_t$, so that

$$s_{t+1} + s_{t-1} = -2\sin(\omega)\, s_t \tag{2.12}$$

Note that we have not made any approximations—this equation is exact. It can be read as two equations, one relating the real parts of the s's, and one the imaginary parts. If we consider only the real parts of the s's, then

$$\mathrm{Re}[s_t] = \mathrm{Re}[i^t z_t] = \mathrm{Re}[i^t(x_t + iy_t)]$$

Thus the real part of s_t for $t = 0$, 1, 2, etc., is x_0, $-y_1$, $-x_2$, y_3, x_4, $-y_5$, etc.; and so equation (2.12), iterated as a real equation, generates consecutive x and y values (with signs sometimes reversed) for points on a circle. If consecutive pairs of points are plotted, for ω small we get a rather good circle (plotting consecutive points like this actually generates an ellipse, but for small ω the eccentricity is very small).[21] If we want to, we can multiply consecutive s_t's by -1 before plotting them whenever $t = 1$ or 2 modulo 4, to correct the signs of the x's and y's, so that points come out in order as we go around the circle.

[21] This evolution has a conserved quantity, which can be derived directly from equation (2.12). Note that

$$s_t(-2\sin(\omega)\, s_{t-1}) = s_{t-1}(-2\sin(\omega)\, s_t)$$

and so, using equation (2.12), we have $s_t(s_t + s_{t-2}) = s_{t-1}(s_{t+1} + s_{t-1})$. From this, with some rearranging, we see that $s_t^2 - s_{t-1}s_{t+1}$ is conserved. If we are at a time step when $s_t = \pm x_t$, then this conserved quantity equals $x_t^2 + y_{t-1}y_{t+1}$, which for small ω is essentially the radius (squared) of the circle.

Now equation (2.12) is of the form (2.3), and so it generates an invertible dynamics. Furthermore, since each point plotted involves two consecutive values of s_t, *each point specifies the complete state of the system.* Thus the x versus y space is the state space for this dynamics, and the dynamical orbits of this system consist of points that all lie approximately along circles, for small ω.

If equation (2.12) is iterated on a digital computer using integer arithmetic, where the product $-2\sin(\omega)\, s_t$ is truncated and rounded off, this equation remains invertible, since the truncation and rounding off will give exactly the same integers when we are going backwards that it did going forwards. Although this discrete evolution is no longer exactly equivalent to the complex exponential we started with, empirically we find that the evolution is still stable, and gives an orbit that approximates a circle.

One is struck by a certain rather strange property of this evolution, which seems less troubling in more abstract contexts. As the pixels light up on our display screen, moving around this circle over and over again, we see a band of a certain thickness develop, until finally the iterated evolution lands on the initial point. Since each point on the screen corresponds to a complete state of our reversible system, the evolution cannot land on any other point twice before it repeats the initial point. Until this happens, when drawing points that are far from our starting point, this evolution tiptoes around points it has already hit, just happening to miss them all. The power of the reversibility constraint to make such an odd series of coincidences happen is rather remarkable to witness.

It is interesting to note that a very similar derivation to the one given above for the Charming Circle leads to a finite-difference version of the Schrödinger equation. In equation (2.10), replace ω by $H\tau$, the Hamiltonian operator times our unit of time τ. Then this equation becomes

$$z_{t+1} = e^{iH\tau} z_t \tag{2.13}$$

In the whole discussion following equation (2.10), we can treat z_t as the wave function.[22] As $\tau \rightarrow 0$, the exact finite-difference equation which corresponds to equation (2.11)

$$z_{t+1} - z_{t-1} = 2i \sin(H\tau)\, z_t$$

turns into the Schrödinger equation, and so the simpler equation in terms of the s's (which has only half as many degrees of freedom, if H is real) can also be considered an exact finite difference version of the Schrödinger equation.

[22] Even the derivation of the invariant goes through—this ends up being essentially the amplitude of the wave function. This invariant can be used to define an inner product which can take the place of the normal inner product in calculations based on the s's.

Chapter 3

Conservation Laws in RCA

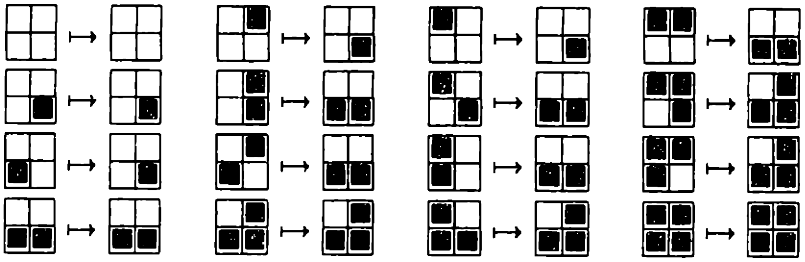
In general, an RCA has as many conserved quantities as there are cells—it *remembers* the initial state of each cell, since you can recover this information by running the system backwards. Thus it is perhaps not surprising that these systems often have invariants which can be computed in a local manner from the current state of the system.

In this chapter, I begin with a survey of invariances in the partitioning rules which are most closely related to the BBMCA. Since this partitioning scheme was invented specifically for the BBMCA, these invariances have not been studied before. This survey will indicate the variety of invariances a small class of reversible rules can have.

I then turn to a number of invariants in second-order RCA. The only previously reported invariants in these RCA were due to myself[45] and Yves Pomeau[59]. I present a number of new invariants which are generalizations of Pomeau's invariants, and show that some of these invariants can be used as generators of the dynamics.

3.1 Invariants in Partitioning RCA

After stressing how reversibility implies conservation, we should of course begin with an example in which we have conservation without reversibility, just to demonstrate that reversibility may be a sufficient condition for conservation, but it isn't a necessary one.



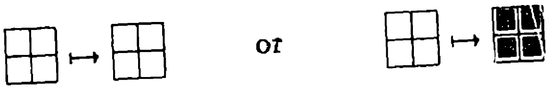
(3.1)

This rule isn't rotationally invariant—we've shown all the cases explicitly. Since each case preserves the number of 1's, this rule is conservative. Since it doesn't always map a distinct initial state of a block onto a distinct final state, this rule isn't reversible. As the corresponding automaton evolved, it would forget all sorts of details about the initial state, but it would always remember the numbers of 1's and 0's. Thus the existence of an interesting local conservation law does *not* depend on the rule being reversible.

We've already seen the BBMCA, which is a reversible rule which conserves 1's and 0's.[1] In this section we'll discuss reversible rules

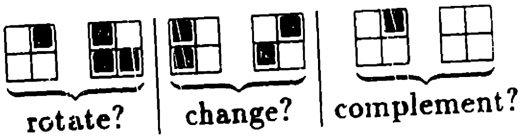
[1] It also has one other conservation that we noted when we discussed this model:

which are closely related to this model. We'll consider all rotationally invariant reversible rules defined on the 2×2 block partition, which are also invariant under inversions. There are 64 such rules: the only operations that can be performed on a block that are rotation and inversion invariant are rotation by 180°, and complementation of all cells. For example, for a block which initially contains all 0's, we must have either



Once we've decided what happens to the all-zeros case, we have no choice about the all-ones case, since these are the only two cases which are invariant under 90° rotations. Interestingly enough, if we impose both conservation of ones and rotational symmetry, then its no longer possible to write down a non-invertible rule. This is why our irreversible example in Table 3.1 had to be given in full!

We can assign a number to each of the 64 possible rules by giving a one-bit answer to each of the questions below for each of the 6 cases indicated.



The middle group could equally well have had the question *rotate?* or *complemcnt?*, since complementation and rotation are the same for

groups of four 1's that are placed as mirrors are *decoupled* from the rest of the evolution, and are permanent. We will not discuss such decoupled invariants for related rules here, but we will mention such situations again in connection with second-order rules in Section 3.2.1.

these cases.[2] Each rule therefore corresponds to a 6-bit binary number. (The rotations are done first, if selected, before complementation).

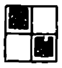
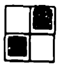
All of the rules that are multiples of 100 (binary) are conservative (and vice versa). These 16 rules include the BBMCA, the identity rule, various gas-dynamics rules (Section 4.2), dynamical spin models (Section 4.3.1), and rules which simulate elastic strings (Section 4.3.2). Adding 11 (binary) to the number corresponding to each of the 16 conservative rules gives another set of 16 rules for which the number of 1's in one step is equal to the number of 0's in the next step (all rules with numbers of the form xxxx11).

The 16 rules that are less than 10,000 (binary) have another very interesting conservation property: they preserve the parity on the unused blockings. Recall that when we use our block rules, we alternate between two partitions; there are two other possible blockings that are never used. Each block of these unused partitions straddles two blocks horizontally in one of the active partitions, and two vertically in the other. If a rule either leaves a block of the active partition unchanged, or complements that block, then the sums of the parities of the two cells along any edge are unchanged—therefore blocks that straddle this active block have the sum modulo 2 of their contents left unchanged. This is a localized conservation—each unused block has its parity preserved for the entire duration of the RCA evolution. In Sections 4.3.1, 4.3.2 and 5.2 we discuss interesting rules that have this conservation.

[2] Each of these cases has only one allowed rotation that changes them.

Note that there are four rules which are both conservative and preserve the parity on the inactive blocks (one of these is the identity rule); and four complementary to these which are 1/0 conservative, and conserve the inactive-block parities.

There are 16 rules (all those with numbers of the form 01xx00, 10xx00, 11xx00, and 11xx11) which have a scale invariance property: we can take any configuration for any of these rules and uniformly spread out all of the blocks of this configuration by inserting blocks containing either all 0's or all 1's (which is needed depends on the rule—those of the form 11xx00 or 11xx11 work with either) between all blocks of the original configuration. This new configuration will then have an evolution which, at regular intervals, is just a spread out version of each step of the original evolution. For example, if we take any BBMCA configuration and make it twice as high and twice as wide by adding one block-column and one block-row of zeros between every block of the original configuration, we get a system that simulates the original system at one half the speed, using four times the area. This happens because the new system may be thought of as being composed of 4×4 blocks: whatever is put in the four corners of these 16-cell blocks propagates into a 2×2 area at the center, where it follows the BBMCA rule. The four cells where the interaction takes place may now be thought of as the corner cells of 4×4 blocks constituting the other partition. The "fragile" BBMCA circuit that we used in Figure 2.9 was constructed by a scaling transformation: scaled mirrors become dynamical objects, which can be destroyed if they are hit at the wrong moment.

Of the conservative rules, there are two rules which conserve linear momentum: the rule in which whatever is in one corner goes into the other corner (the non-interacting gas) and the similar gas rule, in which the head-on collision case causes scattering ( ↦ ), but otherwise everything goes straight through. This rule is equivalent to the HPP gas rule[31], which was the precursor to the current interest in lattice gas dynamics for simulating the Navier Stokes equation. Both of these rules have the property that momentum is separately conserved along every 45° diagonal.

Half of the conservative rules (all the multiples of 1000—these are the rules that are invariant under a 180° rotation) conserve the number of 1's that are on the positive diagonals of blocks separately from the number of 1's on negative diagonals. This conservation will be the basis of our analysis in Section 4.3.2 of a rule that simulates elastic strings.

Rule 10 conserves the overall parity of the entire lattice.

In some of these rules, such as rule 10 and those that are conservative, the invariant is directly a property of the logic gate employed. In the parity-on-unused-blocking conservation, the conservation is a property of the format of the updating. Of these 64 rules (32 if you factor out an equivalence), about a quarter have provided useful physical models, some of which will be discussed in Chapter 4.

3.1.1 Energy in the BBMCA

Since the BBMCA is so closely related to the BBM, in which energy is readily identified, it is tempting to look for an analogy which may be helpful in directing research towards more physics-like models.

In the BBM, the kinetic energy is proportional to the number of moving 1's. In the BBMCA, if we let $p_{x,y,t-1/2} = c_{x,y,t} - c_{x,y,t-1}$, then $\sum_{x,y}(p^2_{x,y,t-1/2}/2)$ counts the number of moving 1's (each moving one disappears from one cell, and appears in another, so $\sum_{x,y} p^2$—which counts how many places change—would count each moving one twice).

The 1's that aren't moving are at those places that were a 1 at $t-1$, and still are 1 at time t. Thus the number of stationary 1's is $\sum_{x,y} c_{x,y,t}c_{x,y,t-1}$. A complicated way of writing the (constant) total number of 1's is

$$E_{t-1/2} = \sum_{x,y} \frac{p^2_{x,y,t-1/2}}{2} + \sum_{x,y} c_t c_{t-1} \tag{3.2}$$

During a collision, some of the *kinetic-energy* changes into *potential-energy*, and then it changes back again.[3]

Since (3.2) is a constant for *any* rule for which $\sum_{x,y} c^2_{x,y,t}$ is constant, it is not possible to derive the particular rule from this expression. We won't pursue the question of using an energy function as the generator of dynamics here; we will have more to say about it in Section 3.3 in

[3] One can think of mechanical models of the BBMCA for which the two terms of (3.2) are proportional to the physical kinetic and potential energy of the system midway between two steps.

terms of second-order RCA. A connection relating this back to partitioning rules will be discussed in Section 4.3.

3.2 Invariants in 2nd-order RCA

In this section we will discuss two kinds of invariants that have been found in second-order RCA. The first discussed are localized invariants: those that you see if you watch the automaton run, even started from a random initial state, since some areas are remaining fixed or going through some short-period cycle. These are perhaps the most obvious invariants. Wc have already seen an example of such an invariant in Figure 2.11. The second kind we will discuss are more energy-like—in fact the first invariant of this sort was discovered in an RCA model which can be used as a dynamical Ising model, and the invariant is the Ising energy.[4]

3.2.1 Localized invariants

In RCA, the simplest locally-computable invariants are of course cells whose values never change. Such situations can arise because many rules ignore the remainder of the neighbors when part of the neighborhood has some particular configuration.[5] For example, consider any

[4] This model, the Q2R model, has been used as the basis of the fastest and largest Ising simulation ever done.[34].

[5] For rules with 2 states per cell, only two rules, "count the parity of the neighborhood" and its complement, have no configuration of part of the neighborhood

rule that, in all cases where the center cell of the neighborhood is 1, ignores the rest of the neighbors and returns a 2. Such a rule, when used in a second-order evolution (equation 2.5), results in a very simple conservation law. If we look at the case in one dimension where the automaton at two consecutive time-steps looks like this:

$$\begin{array}{lll} t-1 & \;.\;.\;.\;\mathbf{1}\;.\;.\;. & \text{A `.' indicates a cell} \\ t & \;.\;.\;.\;\mathbf{1}\;.\;.\;. & \text{whose value is irrelevant} \\ & & \text{to the discussion.} \end{array} \tag{3.3}$$

Any cell which has a value of **1** in two consecutive configurations will *always* be a **1**, regardless of what is happening around it.

For a more interesting 1-dimensional example, consider a 2 state per cell CA with a rule f that returns a 1 iff each of the two cells adjacent to the center is the same as the center:

$$f(c_{\{x\},t}) = \begin{cases} 1 & \text{if } c_{x-1,t} = c_{x,t} = c_{x+1,t} \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

With this rule, a and b standing for any binary values, and $\bar{a}$, $\bar{b}$ their binary complements, the second-order time evolution given by (2.5) says that

$$\left\{ \begin{array}{ll} t-1 & \dots a\bar{a}\dots \\ t & \dots b\bar{b}\dots \end{array} \right\} \rightarrow \left\{ \begin{array}{ll} t & \dots b\bar{b}\dots \\ t+1 & \dots a\bar{a}\dots \end{array} \right\} \tag{3.5}$$

which is again of the same form, so these two cells are decoupled from the rest of the evolution. Any cell which is *not* initially part of such

that makes the remaining neighbors irrelevant.

a pair will never be (and never was);[6] counting all such cells gives us an (invariant) estimate of how many cells are available to represent dynamically changing information (but only an estimate—whole regions may be decoupled from the rest of the evolution because they are surrounded by a wall of decoupled cells;[7] a local counting wouldn't reveal this).

3.2.2 Energy-like invariants

In the preceding section, we discussed some examples of invariants which occur because for certain initial conditions, certain degrees of freedom are decoupled from the rest of the evolution. If we wish to develop strong analogies with physics, we would like to find some invariants which have more of the flavor of an energy in mechanics—quantities that are conserved even in dynamical situations.

We have already seen an example of such an invariant, at the end of Section 2.7.1. In this case, we used a second-order rule to simulate the BBMCA, and for a certain class of initial conditions, this rule has all of the invariants of the BBMCA. Such context sensitive invariants will not be analyzed further here, except to note that invariants arising

[6] In irreversible CA, a guarantee that a cell will always be part of such a pair does not guarantee that it always has been.

[7] An extreme instance of *decoupling* of entire regions occurs with any rule that doesn't depend on the center cell, but depends on its nearest neighbors—this is discussed in Section 4.3. The system decouples into two entirely independent (but interleaved) space-time sublattices, each evolving without reference to the other.

from such embeddings of one rule in the evolution of another are rather common (see Section 4.3.2, for example). The most extreme case is of course a computation universal rule which can be used to simulate any other rule, and hence inherit any set of invariants![8]

Our discussion of more energy-like invariants in second-order RCA will be largely a generalization of an invariant that was discovered by Yves Pomeau[59] for the Q2R rule of Vichniac and Bennett. We will not follow his derivation.

We will base our rules on a slight variant of equation (2.5), given by

$$c_{\vec{x},t+1} = f(c_{\{\vec{x}\},t}) + c_{\vec{x},t-1} \tag{3.6}$$

Note that we have a *plus* sign between the two terms on the right hand side. For rules that employ binary valued cells, this change will not make any difference (since in modulo 2 arithmetic, there is no distinction between plus and minus).

With reference to equation (3.6), the neighborhood $c_{\{\vec{x}\},t}$ consists of some group of cells arranged symmetrically about the cell at position $c_{\vec{x}}$, and f depends only on the neighborhood count—the sum of the values of all neighbor cells. In fact, if $V_{\{\vec{x}\},t}$ is the neighborhood count for the cell at position $\vec{x}$, then all of the rules we will initially be considering

[8] This is a bit like the case of energy conservation in a computer simulation of celestial mechanics—this conservation is entirely a property of the particular simulation, and not an inherent property of the computer.

will be of the form

$$f(c_{\{\vec{x}\},t}) \begin{cases} \neq 0 & \text{if } V_{\{\vec{x}\},t} = k, \\ = 0 & \text{otherwise} \end{cases} \tag{3.7}$$

where k is some integer.

For our first example, we will consider a 1-dimensional CA, with the neighborhood consisting of the two nearest neighbors to the cell at position $\vec{x}$, and $k = 0$. By virtue of equation (3.6), we know that the quantity

$$c_{\vec{x},t+1} - c_{\vec{x},t-1}$$

will equal zero unless $f(c_{\{\vec{x}\},t}) \neq 0$, that is, unless $V_{\{\vec{x}\},t} = 0$. Thus the product

$$(c_{\vec{x},t+1} - c_{\vec{x},t-1})V_{\{\vec{x}\},t} \tag{3.8}$$

is *always* equal to zero, and so

$$\sum_{\vec{x}} c_{\vec{x},t+1} V_{\{\vec{x}\},t} = \sum_{\vec{x}} c_{\vec{x},t-1} V_{\{\vec{x}\},t} \tag{3.9}$$

(since every term is equal). Letting $\hat{a}$ be our unit vector along the lattice, we can write out $V_{\{\vec{x}\},t}$ explicitly, and so

$$\sum_{\vec{x}} c_{\vec{x},t-1} V_{\{\vec{x}\},t} = \sum_{\vec{x}} c_{\vec{x},t-1}(c_{\vec{x}+\hat{a},t} + c_{\vec{x}-\hat{a},t})$$

Every term in this sum involves a product of a cell at time $t-1$ and a cell to one side or the other of it at time t. If we think of each such product term as a *bond*, we get exactly the same bonds from the sum

$$\sum_{\vec{x}} c_{\vec{x},t}(c_{\vec{x}+\hat{a},t-1} + c_{\vec{x}-\hat{a},t-1})$$

and so equation (3.9) becomes

$$\sum_{\vec{x}} c_{\vec{x},t+1} V_{\{\vec{x}\},t} = \sum_{\vec{x}} c_{\vec{x},t} V_{\{\vec{x}\},t-1} \tag{3.10}$$

Thus the quantity $\sum_{\vec{x}} c_{\vec{x},t} V_{\{\vec{x}\},t-1}$, which is just a sum over all bonds, is invariant under this particular second-order evolution.

More generally, let n be the number of neighbors in a neighborhood consisting of some number of pairs of cells, each pair forming the end-points of a line segment bisected by the center cell (the neighborhood may also include the center cell); and let the evolution be governed by any equation of the form (3.7). Now define $c'_{\vec{x},t} = c_{\vec{x},t} - \frac{k}{n}$, and $V'_{\{\vec{x}\},t} = \sum_{\{\vec{x}\}} c'_{\vec{x},t}$ (the sum over the neighborhood of these adjusted cell values): with this definition, of V', when $V_{\{\vec{x}\},t} = k$, then $V'_{\{\vec{x}\},t} = 0$. Also, when $c_{\vec{x},t+1} - c_{\vec{x},t-1} = 0$, we will also have $c'_{\vec{x},t+1} - c'_{\vec{x},t-1} = 0$. Thus we again have

$$\sum_{\vec{x}} c'_{\vec{x},t+1} V'_{\{\vec{x}\},t} = \sum_{\vec{x}} c'_{\vec{x},t} V'_{\{\vec{x}\},t-1} \tag{3.11}$$

Again we'll consider the second sum in this equation. We will consider each pair of cells that are arranged symmetrically about the center cell separately. Each such pair can be re-summed, exactly as we did in the 1-dimensional case, to exchange the roles of t and $t-1$, and so again we find that $\sum_{\vec{x}} c'_{\vec{x},t} V'_{\{\vec{x}\},t-1}$ is an invariant.[9]

This result can be generalized somewhat. First of all, given any rule with a pairwise symmetric neighborhood—not necessarily a rule

[9] Notice that this argument goes through if we use any multiple of c' instead of c'—in detailed calculations it may be convenient to use a multiple of c' in defining the invariant, to avoid dealing with fractions.

of the type given in equation (3.7)—if we can find a linear function V of the neighbors which treats symmetrically opposite neighbors in a symmetric manner, and which is zero whenever f is non-zero (it may be zero at other times as well) then this function can be used in equation (3.8) to make it *always* be zero, and so our whole derivation goes through again. If instead we find a V which treats symmetrically opposite neighbors in an *antisymmetric* manner, then our derivation goes through, except that we get a $(-1)^t$ dependence in our invariant.

For example, suppose we have the following function for f:

$$f(c_{\{\bar{x}\},t}) = \begin{cases} 1 & \text{if all neighbors have the same value} \\ 0 & \text{otherwise} \end{cases} \tag{3.12}$$

Then for V we could use the difference of any pair of symmetrically placed neighbors, and get an invariant with a $(-1)^t$ dependence. If the center cell is included in the neighborhood, we could let V equal the sum of any two symmetrically placed neighbors, minus twice the center cell—this would give an invariant that has no t dependence. The sum of any combination of such invariants, taken with arbitrary weights, is of course also an invariant.

As a further example, consider a two dimensional second order RCA governed by the equation (3.6), employing Boolean variables and a Cartesian lattice. The neighborhood will be the four nearest neighbors, and f will be a 1 whenever exactly two of the neighbors are 1's, but not if the two 1's are opposite each other. Without this extra condition, this rule would be of the form (3.7). However, if V is zero whenever the count is exactly 2, then it is also zero in all cases when f isn't, and so it generates an invariant. If we call the four neighbors N, S, W, and E (North, South, etc.) then $V = N + S - E - W$ also treats symmetric neighbors symmetrically, and is zero in all cases when f isn't—this second V generates another invariant.

Finally, note that if f is a rule which operates on Boolean valued cells and is symmetric between 1's and 0's, and the successive configurations generated by f starting from configurations a_0 and a_1 are called a_2, a_3, etc., then the configurations generated by the rule $\bar{f}$ (the rule that returns a 1 whenever f would return a 0, and vice versa) starting from a_0 and a_1, would be $\bar{a}_2$, $\bar{a}_3$, a_4, a_5, $\bar{a}_6$, $\bar{a}_7$, etc. Since the configurations are the same up to a complementation of all cell values, any invariant of f can be transformed into an invariant of $\bar{f}$.

In all of our examples, there has been a clear connection between symmetries of the rule and the invariants we have been able to construct, which is quite analogous to the situation in physics.

3.3 Hamiltonian dynamics

In the Hamiltonian formulation of mechanics, the variables that are used are those for which the state space of the system evolves like an incompressible flow.

Since our RCA perform a permutation on their set of allowed states, any RCA model evolves like an incompressible fluid in the discrete state space in which each orthogonal dimension corresponds to the allowed values of one of the cells. If you pick any set of points in this state space, there are exactly as many states flowing into this set as out of it.

Thus we are led to try to establish a Hamiltonian analogy for RCA using the cell values as our p's and q's. The discussion of invariants in second-order systems immediately gives us some example systems for which we can generate a dynamics in a local manner from an invariant.

Take any Boolean valued RCA of the form (3.7). For example, consider the Q2R rule which prompted Pomeau's discovery of an invariant in CA: on a 2-dimensional Cartesian lattice, with a neighborhood consisting of the four nearest cells to the center (but not including the center), f is 1 if exactly 2 of the 4 neighbors are 1's, and 0 otherwise. To derive our invariant, we need to take $c' = c - \frac{k}{n} = c - \frac{1}{2}$. Thus our Boolean values become $\pm\frac{1}{2}$. To avoid dealing with fractions, we can use a multiple of c' in defining our invariant, and so we will use ± 1.

Our invariant is just the sum over all bonds between cells in two successive configurations that are one position displaced from one another. It becomes clear now why this quantity is conserved:

When we perform the second-order updating, we look at a neighborhood in one configuration (at time t say), and decide whether or not to flip the center cell in a second configuration $(t-1)$ in order to construct the future $(t+1)$. Although we normally think of the configuration we are constructing as being distinct from the configuration from time $t-1$, we could literally flip the cells in the old configuration whenever the rule returned a 1, and convert it into the new one.

When we've updated all cells, we simply interchange the roles of the two configurations. We can (if we like) do each complete updating step by updating cells one at a time, in any order, since the configuration upon which all decisions are based is not being changed.

Now consider all bonds between the cell we are changing and the configuration upon which the decision is being based. There are only four, to the four neighbors. Our rule is that if exactly two of the neighbors are $+1$, and two are -1 (remember we're using c') then we are supposed to make a change, and otherwise not. But this is the same as saying that we are allowed to change the cell only if it won't change the count of these four bonds (and hence won't change the total count of bonds). We're allowed to make a change only if

$$c_{\bar{x},t-1}V_{\{\bar{x}\},t} = 0$$

both before and after the change to $c_{\bar{x},t-1}$, which happens only when $V_{\{\bar{x}\},t} = 0$.

Thus in this and in all such Boolean valued rules with a single neighborhood count for which $f = 1$, the invariant we have derived can be used to generate the dynamics, using the prescription: *Make any change which doesn't change the "energy,"* and applying it alternately to the two configurations, using the other to provide the neighborhoods.

This prescription can also be applied to other systems for which we've found invariants. For example, consider the system which is the same as the one discussed above (Q2R) with this change: $f = 0$ if the two 1's are exactly opposite from each other. As we pointed out in the previous section, this rule still has the Q2R invariant; it also has a second invariant which, again using N, S, W, and E for the four neighbors, uses a V of the form $V_2 = N + S - E - W$. This together with the first invariant (based on $V_1 = N + S + E + W$) completely characterizes the rule—if both V_1 and V_2 are zero simultaneously, then there must be exactly two $+1$'s and two -1's in the neighborhood, *and* there must not be two 1's opposite from each other.

We can form a single V from these two separate V's as follows:

$$V = \frac{3}{2}V_1 + \frac{1}{2}V_2 = 2(N + S) + (E + W)$$

Since it turns out that both V_1 and V_2 are always powers of 2, multiplying one of them by 3 prevents them from ever cancelling each other—we can only get zero if both are zero. By considering either factors or range it is always easy to combine V_n's in such a manner.[10]

The invariant generated by this V again completely characterizes our rule: the evolution is generated by making whatever changes preserve this invariant. In a similar manner, we can construct invariants which generate the dynamics for a Boolean rule of the form (3.12) which includes the center cell in the neighborhood, but not for one that doesn't include the center.

[10] Notice that the form of V is a sum over symmetric pairs of neighbors, with different weights for each pair, as expected.

Chapter 4

CA models of physics

The use of CA to model physical systems has become a large and rapidly expanding field of research (see for example [10,17,19,20,26,29,31,34,36, 37,46,53,54,65,80,85,86,93] and references therein). I will not attempt to survey this field, but will only give background material when it is necessary for understanding the models or results I am presenting.

I will also not attempt here to duplicate the discussion of physical modeling already given in [80]; instead I will only present a few systems that are of particular interest from a modeling or a conceptual standpoint, based on the partitioning scheme introduced in Section 2.3. Some of these systems are also discussed briefly in Chapter 7, in the course of illustrating the use of various features of a cellular automata machine.

4.1 Constructing CA models

Ideally, one might seek some set of *correspondence* rules, whereby one could start with a physical Hamiltonian and transform it into a CA rule. This is to some extent the inverse problem of the one discussed in Section 3.3, where we found rules which have invariants which can be used to generate the dynamics—further progress in this direction may lead to useful general techniques.

In the meantime, the most productive technique for constructing physical models is to build upon known models (such as the lattice gas models[31,26]) by adding extra states that modify the dynamical interaction, or by coupling the dynamics of two well-understood systems, each of which uses part of each cell's state. To find completely original models, it often helps to start with the symmetries and conservations that the macroscopic dynamics is to have, and try to put them into a simplified microdynamics (i.e., a rule).

The most straightforward application of these techniques is in the context of *partitioning* models (see Section 2.3): the state variables are partitioned into disjoint subsets, each of which is updated as a group; then the partition is changed.

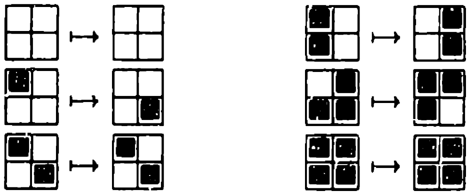
The partitioning technique is easy to implement (see Section 7.6.2) and allows a dynamical evolution to be specified in a particularly direct manner: we give a series of *before* and *after* configurations for groups of state variables. Since all variables involved in each group see exactly the same neighborhood information, its easy to get them to act in a coordinated fashion. Properties of the local dynamics, such as invariances and symmetries, are inherited by the global dynamics in a rather straightforward manner, as we have discussed in Section 3.1.

Note that the input partition and output partition for a given update step need not be identical; such a situation will be termed a *shifting* partition. An update step using a shifting partition is equivalent to a step using an ordinary, *fixed* partition followed by a step in which state variables are translated to new positions.

Except where noted below, our examples in this chapter will all involve an alternation between two fixed partitions: the two 2×2 block partitions used for the BBMCA rule of Section 2.4.

4.2 Models based on lattice gases

One of the conservations that can be put into a partitioning rule "by hand" is conservation of momentum. Consider the following rule



(4.1)

(where again *rotated case* ↦ *corresponding rotation*). In all cases, whatever is in one corner moves to the opposite corner. The global evolution that is generated by this rule is rather trivial: all 1's and 0's travel in straight lines, and never interact at all. If we assign some momentum to the travelling 1's, then this rule clearly conserves mo-

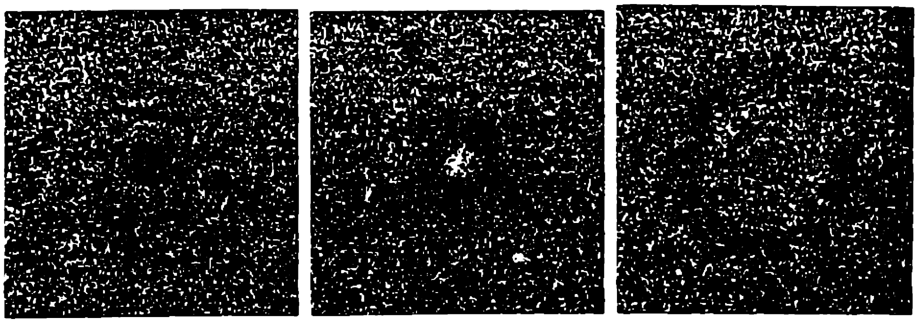
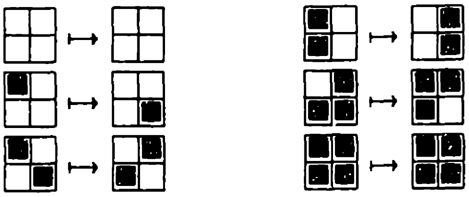


Figure 4.1: Circular wave produced by a localized disturbance.

mentum, but is rather unexciting. Suppose we now change the third case, so that the rule becomes



(4.2)

Now if two 1's (or two 0's) approach each other head-on, the "particles" come out rotated by 90°. This turns our previously non-interacting gas into an interacting gas, with momentum conserving collisions. Figure 4.1 shows 3 frames from the evolution of this rule, started frcm a uniform 50% density of 1's and 0's, except for a 16×16 area in the middle, where the density was 100%.

This rule, along with other gas rules, was discovered as a natural extension of the techniques de·veloped for the BBMCA model. It turns

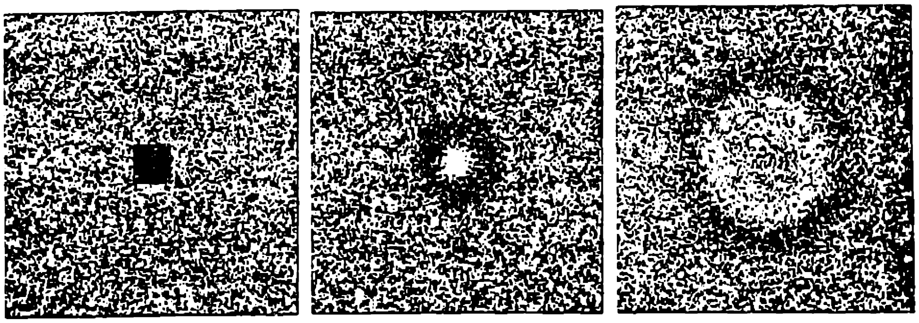


Figure 4.2: Higher resolution version of the circular wave; only sites containing 3 or more particles are colored.

out to be equivalent to a lattice gas model (the HPP model) developed more than a decade ago by Yves Pomeau[31] and coworkers. His original model was given as the alternation of two rules on 4-bit cells using only nearest neighbors which happens to be equivalent to a shifting partition. When we implement it this way on CAM-6 (see Figure 4.2) we get more sites into a 256×256 area (the size of the display screen), but this implementation has the disadvantage that the system shown decouples into two independent sublattices, a phenomenon we will discuss in Section 4.3.

4.2.1 A gas with a finite impact parameter

Although the HPP model depicted above spawned later improved lattice gas models (such as the FHP model based on a triangular lattice) which reproduce the Navier Stokes dynamics in the macroscopic limit[26], the original model has significant defects. One defect is the fact that this rule conserves momentum separately on each diagonal; this makes it behave in some ways more like a 1-dimensional system than a 2-dimensional system. In Section 7.4 (and in [46]) we describe an experiment where Toffoli, Vichniac, and I used CAM-6 to measure velocity/velocity time-autocorrelations at a given site

$$\nu(t) = \lim_{T\to\infty} \frac{1}{TNQ} \sum_{t'=1}^{T} [a^q_{ij}(\phi^{t+t'}C) - \bar{a}][a^q_{ij}(\phi^{t'}C) - \bar{a}]$$

where C is an initial configuration, ϕ is the transition rule, $a^q_{ij}(C)$ has a value of 1 or 0 depending on whether or not there is a particle moving in the q direction at the (ij) site in the configuration C, $\bar{a}$ is the average particle occupancy (per site) for each direction, and there is an implied summation over all N sites and all Q allowed directions. The measured results for three lattice gas models are given in Figure 7.2. The asymptotic slope for the HPP gas is that of a 1-dimensional system[58]; the expected slope for a 2-dimensional gas is -1, which is clearly exhibited by the gas labeled TM on the graph.

This TM (Toffoli/Margolus) model can be described in terms of the rule of Table 4.2: follow the HPP rule, except that at even time steps, rotate the result for each case 90° clockwise, and at odd times rotate the results 90° counterclockwise. This rule is almost a simple when described directly: take the initial state of a block and rotate it 90° counterclockwise on even steps, and clockwise on odd steps, unless it contains exactly two 1's on a diagonal, in which case the block is left

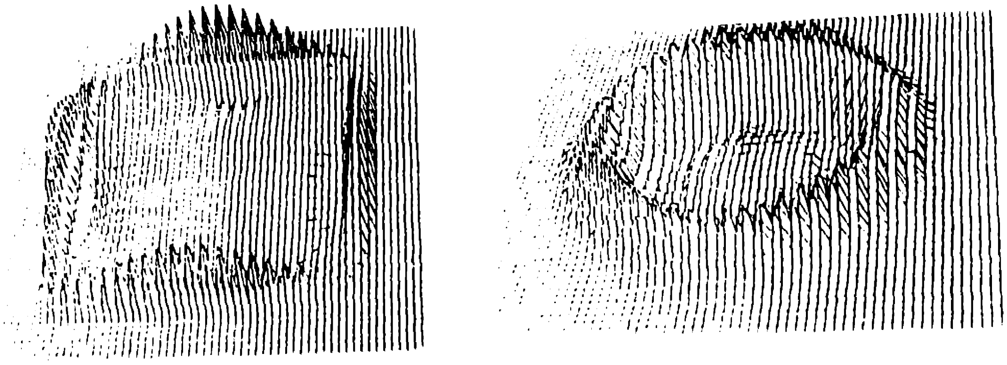
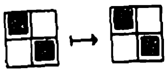


Figure 4.3: Probability flows in the HPP and TM gases, respectively

unchanged



This rule causes particles to travel in straight lines vertically and horizontally; particles on adjacent rows or columns can collide, moving them from two adjacent rows to two adjacent columns, or vice versa. Thus collisions in this rule have a finite impact parameter, unlike the HPP collisions. This allows momentum on different rows and columns to interact, avoiding the spurious conservation seen in the HPP model. In Figure 4.3 we show the results of a floating point simulation (conducted by Tom Cloney using a Lisp machine) which shows the average response of each of these gases to a (probabilistically) very small perturbation in density at a single site.

This technique for constructing models with a non-zero impact parameter may be useful in other lattice gas models.

4.2.2 Reflection and refraction

All of the models mentioned above support the wave equation (in terms of particle densities). We can therefore modify these models in such a way as to produce reflection and refraction phenomena in these waves.

By adding an extra bit to each cell's state (which doesn't change with time), we can mark areas where we wish to follow a different rule. If in all blocks in which any cell is so marked we follow the *identity* rule (nothing changes) then such marked areas will act as mirrors—any particle that hits such an area will bounce back the way it came. In Figure 7.5 we show a wave colliding with a concave mirror formed in this manner.

Such reflection phenomena using these gases are of course seen whenever these gases are modified to accomodate obstacles. A more original modification is to have marked regions follow the original rule some portion (say half) of the time, and the identity rule the other half of the time. This results in marked regions having a "sound" velocity that is half that of unmarked regions. In Figure 7.6 we show the refraction of a *soliton* by such a region—we used these high-density waves supported by our rules (TM or HPP) in order to make the refraction more evident in this rather low-resolution picture.

This technique may be a useful one for implementing other kinds of potentials as well.

4.3 Matched pairs

In this section I would like to discuss some pairs of closely related RCA rules, one of which is a second-order rule, the other of which is a partitioning rule.

In Section 2.7.1, we gave an illustration of how a second-order rule can be constructed which simulates a block rule for some subset of its allowed initial states; here the connection between a block rule and a second-order counterpart will be even closer, and it will be the initial states for the block rule which must be constrained to get an equivalent evolution.

For two second-order rules which provide useful models of physical systems, we will discuss the effect on the "physics" of the situation of relaxing the constraint on the block rule which makes the two rules equivalent.

Let us begin by considering any 1-dimensional second-order RCA rule that depends on the values of the two nearest neighbors, but not on the center cell. A portion of the evolution might look like this

```
t-1          . . . 1 . 1 . 0 . 1 . . .
t            . . 1 . 0 . 0 . 1 . 1 . .
t+1          . . . ? . ? . ? . ? . . .
```

where a dot indicates a cell whose value is irrelevant to the discussion. The cell values that are indicated explicitly are sufficient to allow our second-order evolution to compute the states of the cells marked with '?'. Note that the system decouples into two entirely independent (but interleaved) space-time sublattices, each evolving without reference to the other.

In such an evolution, we can eliminate one of the sublattices, and merge information from pairs of consecutive configurations into single configurations—the unused cells from step t can be used to hold the information from step $t - 1$

```
t-1,t          . . 1 1 0 1 0 0 1 1 1 . .
t,t+1          . . 1 ? 0 ? 0 ? 1 ? 1 . .
```

We alternately update *even* and *odd* sites, to get a first-order evolution isomorphic to the original.

Similarly, on a 2-dimensional Cartesian lattice, a rule that doesn't depend on the center cell but does depend on the four nearest neighbors has an evolution that decouples into two interleaved space-time checkerboards.[1] As before, we can eliminate one of the sublattices by merging information from consecutive configurations. If we think of space as a red and black checkerboard, then we alternately update red cells and black cells.

In Figure 4.4a we show a 2-dimensional checkerboard, with sites

[1] In any number of dimensions, any rule (first or second order) that doesn't depend on the center cell, and only depends on a single pair of symmetrically placed neighbors in each orthogonal direction has an evolution that decouples into interleaved space-time sublattices.

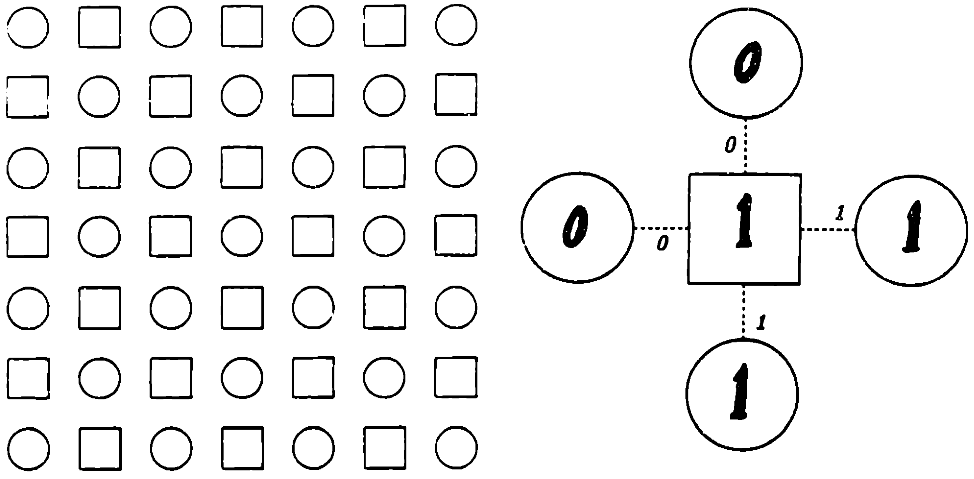


Figure 4.4: Checkerboard. Sites updated on even steps are circles, odd step sites are squares.

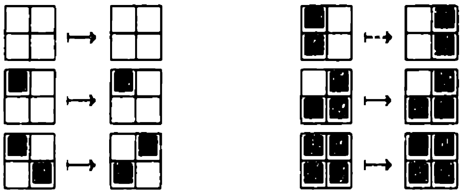
that are updated on even steps shown as white circles, and sites that are updated at odd steps shown as white squares. In Figure 4.4b, we show a closeup view of an odd site that is about to be updated as a function of its even neighbors. The four dotted lines connecting the odd site to its four even neighbors can be thought of as *bonds*, the values written on the bonds indicate whether (1) or not (0) the center is the same as that neighbor. The updating will either leave the center unchanged, or complement it. If the center is complemented, all four bonds are complemented.

If we just consider the dynamics of the bonds, we see that blocks of four bonds are either all complemented, or all left unchanged. As we switch from an even step to an odd step, bonds that were updated as a group during one step are updated as part of four separate groups during the next step. Thus this system corresponds to a 2×2 block partition at 45° to the orientation of the picture. Any second-order rule (using binary-valued cells) that treats 1's and 0's symmetrically and only involves the four nearest neighbors can be recast as a rule on the 2×2 block partition that, for every case, either complements the block or leaves the block unchanged. As we have explained in Section 3.1, all such rules conserve the block parities on the unused partitions—the significance of this will be discussed in our examples.

There are 16 rules of this kind that are rotationally symmetric (256 without this restriction). If we factor out the fact that f and $\bar{f}$ generate equivalent evolutions, there are only 8 distinct rules of this kind. One of these is the identity rule—we will discuss the use of two of the others as physical models.

4.3.1 Dynamical spin models

It was Michael Creutz[17] who first investigated RCA rules suitable for use in Ising simulations. His simplest first-order rule was intermediate between a partitioning rule and a second-order rule—it was in fact a checkerboard updating scheme involving four neighbors, of the sort we have discussed above. In this scheme, we think of the two cell-states as being $+1$ and -1, or spin-up and spin-down. The bonds are the negatives of the products of two adjacent spins: -1 if the two spins are equal, $+1$ if they aren't. The rule for updating is: flip the center spin if doing so doesn't change the sum of the four bonds connected to that spin.[2] As a second-order rule, this is the Q2R rule that was later discovered by Vichniac and shown to be an Ising system by Pomeau, and which has been used for massive Ising simulations by Herrmann[34]. This rule was discussed in the previous chapter, as one which has an evolution which can be generated from an energy function. As a block rule, it has the following table (where positive and negative bonds are represented by ■ and □ respectively)



(4.3)

Blocks are complemented if they contain exactly two ■ 's, and left unchanged otherwise.

This rule has two interesting features, which are worth pointing out. First of all, it conserves the number of ■ 's; this corresponds exactly to the conservation of bond energy. In fact, this rule can be viewed as a dynamics for the *energy*, as opposed to a dynamics for the spins.

Secondly, this rule is more than simply an Ising system. Notice that in our derivation of the correspondence between the checkerboard dynamics and this one, we introduced a variable for every *bond.* But bonds are twice as numerous as the cells in the checkerboard. In the

[2] He was mainly concerned with rules that involved a small energy reservoir (represented by a few bits) associated with each spin. With a reservoir of 1 or more bits, the rule becomes: flip a spin if it can be done without changing the total energy associated with the spin (including energy in the associated reservoir, which may be moved to or from the bonds if necessary).

checkerboard system, if we take the sum around any block of four adjacent cells of the bond variables connecting them, we must always get zero. For our block rule, this is equivalent to requiring that the blocks on the unused partitions must all contain an even number of ■ 's. Only when this condition is satisfied can this block rule correspond to a checkerboard Ising dynamics.[3] On a periodic lattice, once half of the cells have been assigned values freely,[4] our constraint tells us how to fill in the rest. To transform to a checkerboard system, we must specify the state of one spin, and then all other spin values can be determined from the bond information. This one-bit ambiguity corresponds to the symmetry in the energy between a given configuration of spins and the complementary configuration.

If, in the checkerboard dynamics, we somehow add an extra variable on each bond, specifying whether it is *ferromagnetic* or *antiferromagnetic*, then we have a spin glass model. In this case, there is no constraint on the sum of the bonds connecting four adjacent spins. Thus the unconstrained 2×2 block model of Table 4.3 can be interpreted as a dynamical spin glass model, again following a dynamics which "flips a spin" (i.e., complements four bond energies) whenever this doesn't change the total energy.

To a given configuration of bond energies, *any* configuration of spin

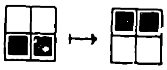
[3] For a system with periodic boundaries, we would also have the condition that any horizontal or vertical path that closes on itself must contain an even number of ■ 's.

[4] Pick any set of cells that can be filled without being limited by the constraints.

values may be associated, by choosing the bond types to be ferromagnetic or antiferromagnetic, as appropriate. The subsequent dynamics of *all* of these systems (spins plus associated bond-types) will be isomorphic from an energetic point of view. Thus the block rule manages to simulate a deterministic spin-glass dynamics in a manner that "factors out" irrelevant details about spin orientations—such a rule uses only 2/3 as many state variables as an equivalent dynamical simulation which keeps track of the spin values[11].

4.3.2 Elastic strings

In Section 3.3 we discussed a rule that is closely related to the Q2R "spin" rule discussed above. In terms of the checkerboard realization, this rule is the same as the dynamical Ising rule, except that if two diagonally opposite neighbors are 1's, the center spin isn't flipped. If we only list the cases which change, the block rule corresponding to this is simply



(4.4)

(and rotations). Empirically it is found that if we start this rule from any configuration involving a single chain of cells that has no slope steeper than 45°, and which obeys the parity constraint on the unused blocks that is needed to make this rule correspond to its second-order counterpart (as discussed in the previous section) we get wave propagation on the string. For example, after about 60 steps of evolution under this rule, the initial configuration of Figure 4.5a goes into the

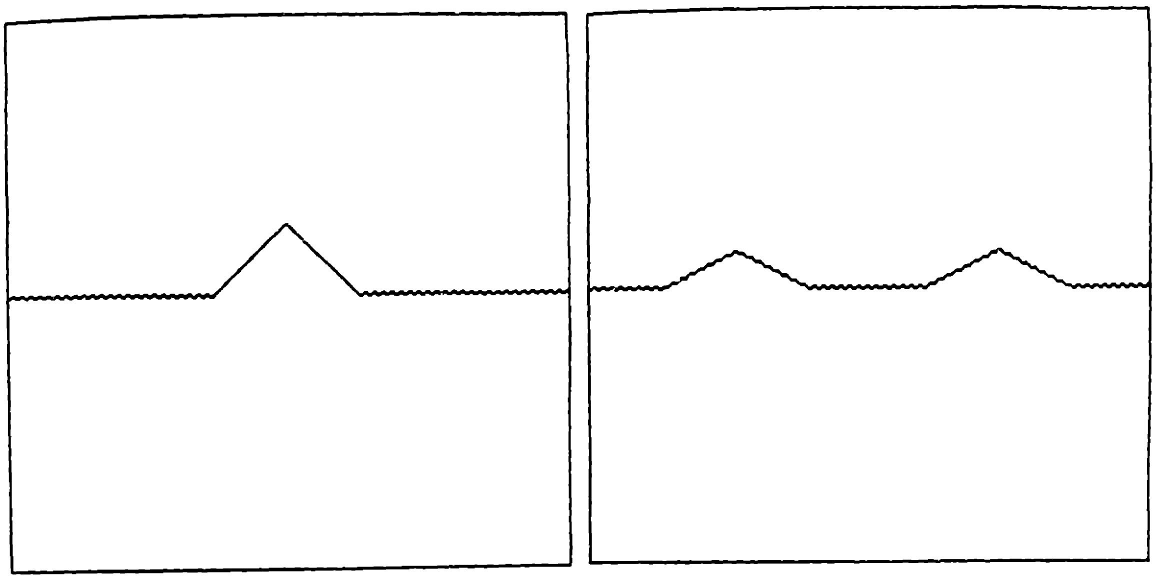


Figure 4.5: Wave propagation. In (a) we have the initial configuration, and in (b) the state after about 60 steps.

configuration shown in Figure 4.5b.

There is a surprisingly simple proof that waves on this string obey the ordinary linear wave equation—this proof is based on an invariant of this rule. As mentioned in Section 3.1, this rule separately conserves the total number of 1's on all positive diagonals of blocks,[5] and the number of 1's on negative diagonals (the one case that changes preserves these counts). This was already evident in the second-order version of this rule, where the quantity

$$\sum_{\text{all cells}} C_{t-1}(N_t + S_t + E_t + W_t)$$

is conserved, where C_t stands for the value of the center cell at time t,

[5] A cell that is on the positive diagonal of a block in one of the active partitions is also on a positive diagonal in the other active partition.

etc., and also the quantity

$$\sum_{\text{all cells}} C_{t-1}(N_t + S_t - E_t - W_t)$$

When we go to the checkerboard version of the rule, we can drop the time subscripts. Adding and subtracting these two invariants, we see that $\sum C(N+S)$ and $\sum C(E+W)$ are separately conserved, which in the block version just becomes the invariant we've already noticed.

If we draw a 1 that is on the positive diagonal of a block of the active partition as ⧅ , and similarly with the negative diagonal and ⧄ , then the total number of ⧅ 's and the total number of ⧄ 's are conserved separately. The reason we've used what might seem a contrary choice of symbols has to do with our parity constraint: blocks on the unused partitions must have a parity of 0 in order for this rule to be equivalent to its second-order counterpart. As a consequence, if we are given a sequence such as "\\\/\//\//," there is only one way in which a chain of 1's can be layed down (from left to right) to occupy the diagonals required by this sequence—just lay out the ⧅ 's and ⧄ 's so that you get an unbroken line (see Figure 4.6).

We are now in a position to prove that our dynamics is that of a linear wave equation. What we will show is that, given a configuration corresponding to a sequence such as "\\\/\//\//," our dynamics carries all of the even numbered elements of the sequence one way, and all of the odd numbered elements the other way. Thus the configuration that we see at any moment is a superposition of a wave travelling to the right, and a wave travelling to the left, and these two waves may

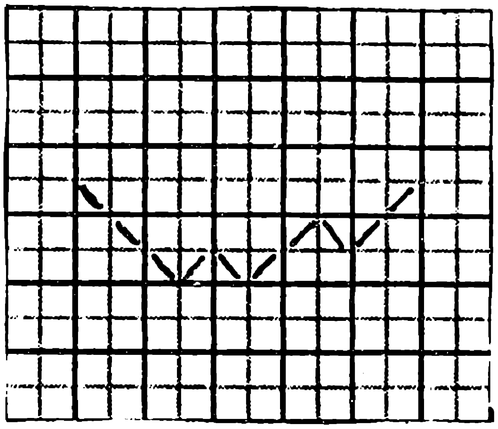
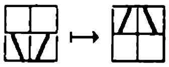


Figure 4.6: String corresponding to the sequence "\\\/\//\//."

be of any shape.

In terms of our ⧅ 's and ⧄ 's, our rule is



(and rotations), all other cases remain unchanged. Now let us follow the rightward progress of a ⧅ started in the lower left corner of a block. On our string, only two cases can arise (because of our parity constraint): if the block contains two 1's side by side (⧅⧄) then our ⧅ moves up and to the right (⧄⧅). If our block contains only a single 1, (⧅) then the cell below and to the right must contain a single 1, since this is the only way that our string can be continued without violating our constraints. This 1 is again a ⧅, and so we may imagine that these two 1's interchange, so that the ⧅ we are following moves down and to the right. In either case, the next updating step (on the other partition)

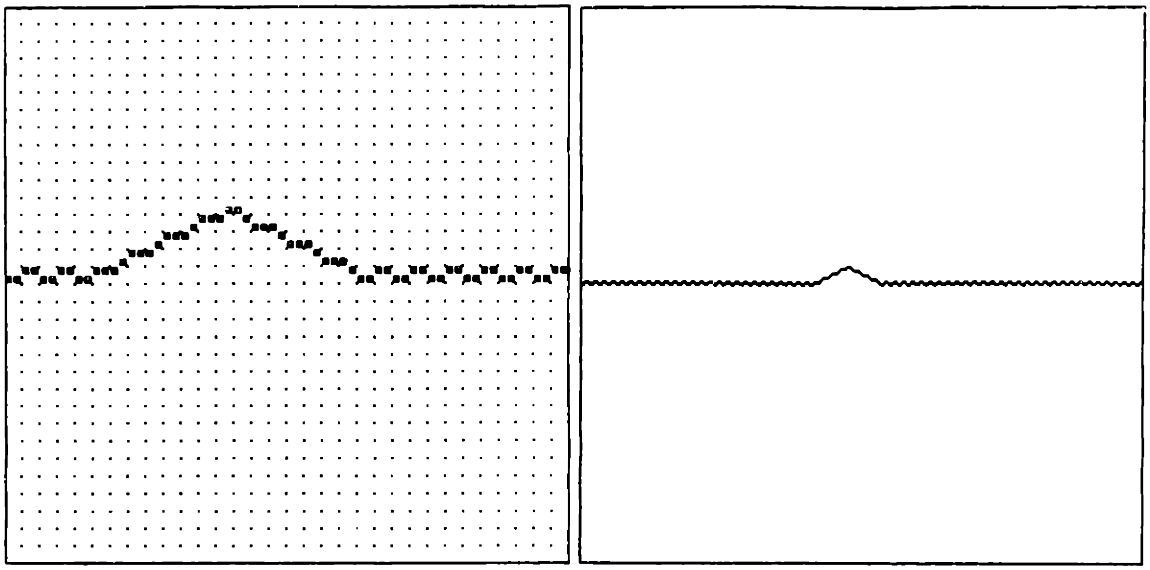


Figure 4.7: In (a) we have a closeup of part of the right-going wave shown in (b).

will again find the ⧅ that we are following in the lower left corner of a block.

In a similar manner, we can see that a ⧄ started in the upper left corner of a block moves one position to the right along our string at each step; similarly 1's started in the right half of a block move left. Thus our evolution is equivalent to a 1-dimensional rule acting on strings of ⧅'s, ⧄'s, and 0's, which alternately interchanges the contents of even pairs and odd pairs of cells.

In Figure 4.7a, we show a wave that we have constructed. The right-going sequence is

```
/\/\////////\\\\\\\\/\/
```

which yields a wave with a peak. The left-going sequence is

```
\/\/\/\/\/\/\/\/\/\/\/\
```

which yields the flattest possible pattern. When we interleave the elements of these two sequences, we get

```
/\\//\\//\///\///\///\//\\\/\\\/\\\/\\\//\\//\
```

which was then transcribed into the chain shown in Figure 4.7a.

Our analysis so far is sufficient for strings that are periodic in space, such as the one used in Figure 4.5. With the block rule we have been using, if a string has ends, then these ends remain essentially fixed: each end can move vertically between two cells (a dangling end results in blocks on the unused blocking that have odd parity, and so this case won't correspond to anything in the second-order rule). When our rightward moving wave hits such a fixed end, it is reflected. In detail, each element of the right moving sequence will be "stuck" for one step when it reaches the right end, and then will begin travelling leftward. Since the rightmost element of the arriving sequence becomes the leftmost element of the departing sequence, when a sequence such as

```
/\/\////////\\\\\\\\/\/
```

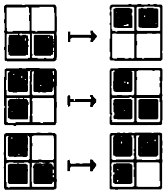
(a peak) arrives, this will become the sequence

/\/\\\\\\\\////////\/\/

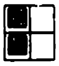
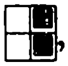
which is a trough. Thus waves reflected from a fixed end are inverted.

In the course of this discussion, you may have noticed something: the constraint of *even* inactive-block parities means that no active block can ever contain exactly two 1's on a diagonal in any of our strings. This means that everything we've discussed so far applies to our spin model of the previous section just as much as to our "elastic rule" introduced in this section! If the waves depicted in Figures 4.5 and 4.7 are used as initial states for our spin model, then they will run as described here.

If we want to have ends that are free to move vertically, we can modify our rule (4.5) so that it becomes



(4.5)

(and 180° rotations). All other cases remain unchanged. Since our horizontal strings have not made use of the cases involving  and , we have let these remain fixed, so that vertical bars remain stationary. We've added 2 cases which allow a 1 next to a vertical bar to move freely along the bar. Note that these cases break the separate conservation of total 1's along positive and negative diagonals: when a ⧄ arrives next to a bar, it becomes a ⧅, and vice versa. Thus both the order of 1's and the kind of 1 are changed, and so a peak such as

```
/\/\////////\\\\\\\\/\/
```

moving to the right becomes

```
\/\////////\\\\\\\\/\/\
```

(another peak) moving to the left.

These string models have been analyzed entirely in logical terms; this analysis has the virtue that it is *exact.*[6] Unlike the analysis of a real *macroscopic* string, there was no opportunity or need to make use of statistical mechanical methods. By combining gas models with models related to those discussed here (which can be used to construct elastic membranes) we arrive at models with statistical potentials.

4.4 Logic

In this section, we will discuss an irreversible logic model, which was designed to show that we can have a partitioning model which can perform logic much more compactly than the BBMCA. This model also illustrates the rather direct way in which partitioning allows a desired dynamics to be transcribed into a CA rule by providing a sequence of before/after pictures of a small patch of space.

There are plans to actually build chips that implement this model in parallel hardware, and can have cell configurations which will simulate

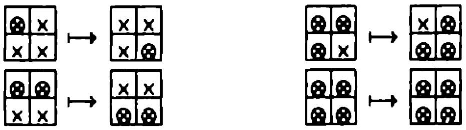
[6] For this reason, this model may be of some pedagogic interest.

circuits downloaded to them. This hardware may be able, in some cases, to simulate these circuits fast enough to allow this chip to actually be used *in place of* the hardware that is being simulated (voltage levels at some input pins would be converted into cell values, and voltage levels at other output pins would reflect certain cell values). Thus although this is the furthest of my examples from microscopic physics, it is perhaps the closest to being realized as a physical system.

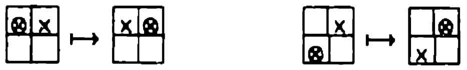
This model makes use of four-state cells—if you think of each cell as containing two bits of data, one of the bits can be thought of as containing *wire* data, and the other *signal* data. A cell that contains neither wires nor signals will be drawn empty; × will indicate a cell containing only a wire; ○ indicates only a signal; and ⊛ indicates both a signal and a wire.

The rule for wires is very simple: they never move or change, regardless of what's around them. The rule for signals that aren't on wires is also very simple: they never move or change (they are only used to influence the behavior of other signals passing nearby).

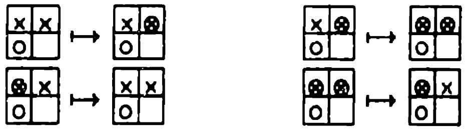
Thus we only need to illustrate what happens in the cases where there is at least one signal and one wire. The rule is rotationally invariant and inversion symmetric. For cells that contain four "pieces" of wire, any signals just go straight through



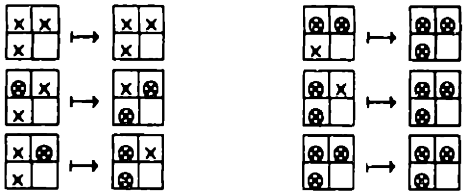
If there are two pieces of wire, signals on the wire follow the wire;



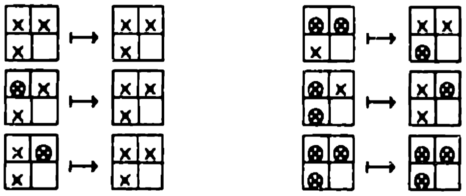
and signals next to a horizontal or vertical wire cause the signal state of the adjacent cell to be complemented before it moves



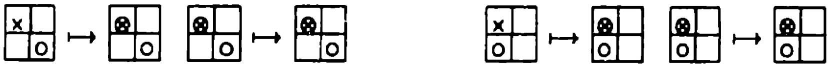
(signals next to a diagonal piece of wire have no effect). If there are three pieces of wire and no nearby signal, the signal value on each wire becomes the logical OR of the signals on the other two wires:



If there are three pieces of wire and there *is* a signal on the remaining cell, the signal value on each wire becomes the logical AND of the signals on the other two wires:



Finally, if there is only a single wire in the block, there will be a signal if there is a signal nearby, but not otherwise:



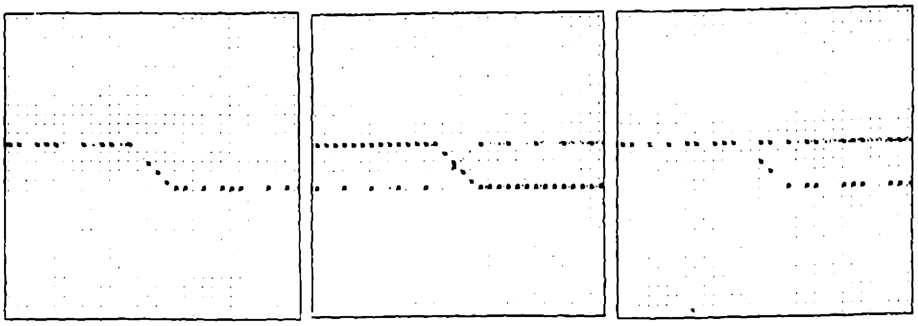


Figure 4.8: Signal propagation, cross-over, and fanout.

Now we simply draw the wires for a circuit, and put signals on them. In Figure 4.8 we illustrate propagation, crossover, and fanout. Notice that signals just follow wires, and signal streams simply cross without influencing each other wherever two wires cross (this is due to the four-wires case above). To achieve fanout, we just have a wire split (being careful to make sure that some block has three wires in it, so that we use the OR case described above).

Figure 4.9 shows configurations that perform NOT and AND operations, and the last frame shows a binary half-adder consisting of about half a dozen gates in an area of about four 2×2 blocks (illustrating that we can achieve quite dense circuitry).

This logic rule could be implemented using unclocked, asynchronous logic using the techniques of Section 5.2 in the next chapter.

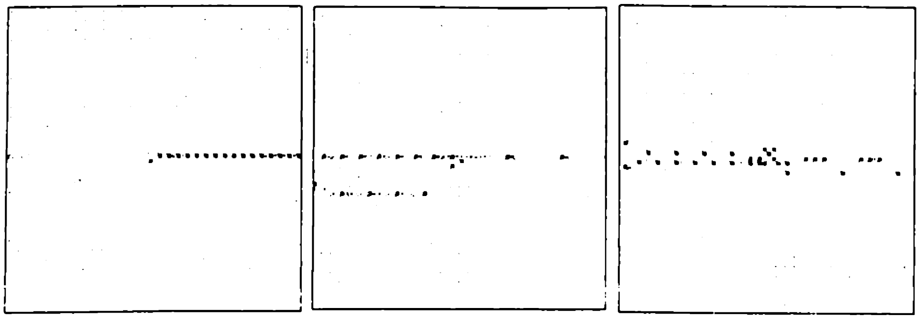


Figure 4.9: A NOT gate, an AND gate, and a half-adder.

Chapter 5

Time and Spacetime

This chapter deals with several aspects of the idea of time in cellular automata. There is of course the normal synchronous time that is usually used to define these models (all cells are updated *simultaneously*). We will also talk about asynchronous models, in which each cell runs as fast as it can, and only waits when a neighbor gets behind, in order to simulate a synchronous updating. We present a simple new scheme for achieving such local synchronization that is made possible by the block partitioning, and which will be used in Chapter 6.

Our discussion of local synchronization will lead us to "relativistic" cellular automata models, in which our RCA can run its evolution on different spacelike slices. But first we will talk about the statistical-mechanical notion of the macroscopic arrow of time in RCA models, which is quite distinct from the time used by the simulator to perform the updating.

5.1 The arrow of time

In microscopic physics, the laws seem to be perfectly reversible, and there is no important difference between past and future; yet macroscopically a difference exists. RCA provide a context of a dynamical system in which the law is known perfectly, in which this same dichotomy can be examined.

5.1.1 Coming full circle

Let us think about a large but finite RCA (with periodic boundaries, say). Being a finite and deterministic digital system, it has only a finite number of possible states, and so it must eventually get into a state that it has been in before. Since it is deterministic, it must then do exactly what it did the first time it went through that state, and so it must cycle. Since it is a reversible system, each state has a unique predecessor, and so given some initial configuration, our automaton cannot repeat any successor to this configuration without passing through the given initial configuration. This behavior should be contrasted with that exhibited by irreversible CA. Such CA may enter a cycle at any point in time—they do not obey a constraint comparable to that which exists for RCA. This explains the empirical observation that RCA typically have much longer periods than non-reversible CA.

Now suppose our rule for our RCA happens to be a computation-universal one, which allows a rich evolution including the evolution of intelligent beings (computers if you prefer).[1] We can imagine the low-entropy[2] initial state gradually becoming increasingly complex, and eventually beings arise and perhaps write books. As entropy continues to increase, the last surviving work of the noted author A falls to dust. The universe eventually becomes random looking, and all traces of structure and purpose vanish. This randomness persists for an unimaginably long time, but not forever. Since our RCA is finite, it must eventually cycle.

How does it cycle? Does it just suddenly fluctuate back into its initial configuration?

There is a very simple way to see exactly what happens: starting from our given initial state, run our reversible rule backwards. Of course this is also an RCA rule[80]. For initial conditions which aren't time symmetric (very few are) we get a completely different evolution, but again from a low entropy start. Again we expect to get complex structures, and even beings, but completely different individuals and

[1] Von Neumann first proposed Cellular Automata as models in which, in principle, life could arise through the operation of simple mechanical laws. He found CA which supported patterns of states that could act as general-purpose computers, and which could make copies of themselves.

[2] In this section, by *entropy* we mean some coarse-grained entropy such as those discussed in Section 2.8.1.

situations from what we got going the other way. Finally the last surviving work of the great author B falls to dust, and we eventually arrive at our high entropy state from this direction in time. So we see that the way that the RCA gets back to its initial configuration from the high-entropy state is by following in reverse an evolution that is just as rich as the initial evolution!

To try to appreciate how strange this is, let us continue to trace this second evolution through the high-entropy phase. For endless ages, the randomness persists, but eventually we see the last surviving work of A emerge out of the chaos! This is quite surprising, since A has not yet lived, following the evolution from the given initial state in this direction in time. We have created a book written by A in a very indirect manner—by evolving a different universe, and then tracing randomness for a very long time! As we continue, we see A unwrite his books, and his whole universe unevolve, and finally we get back to our initial state.

5.1.2 The order of events

Has the entropy of the universe decreased in this second phase of the computation? Clearly it hasn't as seen from A's point of view—the set of states that constitute A's evolution is the same when computed in either direction in time. We just saw the states in the reverse order the second time. A's perception of events cannot depend upon the order in which *we* see these states—nothing in the states themselves contains any information about which order we, on the outside, see them in. From A's point of view, the arrow of time goes from the simple initial state to the high-entropy middle phase. There is a very sharp distinction between the direction in which we compute the states, and the direction in which time flows for beings living in the RCA.

I would refer to the arrow of time relevant to the experience of beings living in the RCA as the arrow of macro-causality. In order to make predictions about the future behavior of the world based only upon the observable macroscopic parameters, one would like to treat the unobservable microscopic variables as uncorrelated, so that the behavior of an average member of the ensemble consistent with the macro-constraints may be assumed. This statistical assumption only has predictive power when going from low to higher entropy—in order to go the other way, non-neglectable correlations must be assumed. Thus it is entropy itself which defines the arrow of time for macroscopic phenomena, and so entropy is always observed (by those inside the automaton) to increase, by definition.[3]

[3] This assumes that the direction of entropy increase is the same everywhere in the automaton at once. For certain initial conditions and certain types of rules, this is not the case. It seems, though, that all such discrepancies are unobservable from a given location within the automaton.

5.2 Synchronous causality

Thus far all of our CA models have been based on the Newtonian notion of a synchronous moment of time during which all cells everywhere are updated simultaneously. In this section we will discuss the problem of having systems which don't have any global clock simulate ordinary synchronous cellular automata. What such systems must do is preserve the *causal* structure of a synchronous evolution: the functional dependence of a cell on the past states of other cells must be preserved, even if the relative timing of updating is changed. If a cell is temporarily ahead of the others, it must have the state that it would have had if all other cells had kept step with it. Though timing will not be deterministic, at each place we want the right events to happen in the right sequence.

5.2.1 CA which are effectively synchronous

Implicit in the usual definition of cellular automata is the requirement that each step of updating should be equivalent to a simultaneous application of the transition rule to all sites.

Since we are unable to build machines which achieve perfect synchronization of activity, we discretize time in order to achieve effectively perfect synchronization. The usual approach is this: all cells are updated once, in a manner that doesn't depend on the order of updating, and then we repeat this process.

For example, we might use two copies of our system, the old system and the new system. We look at the neighborhoods in the old system, and construct the results in the new system. When we have completely constructed the new system, we no longer need the data contained in the old system, and so we can interchange the roles of the two systems, and begin a new updating step.

Each complete updating is a unit of CA time: the actual physical time interval between consecutive steps is irrelevant—whether it is a picosecond or a year, one unit of time has passed in our automaton. The progression of such CA time-steps is analogous to the progression of months—it is January for everyone before it is February for anyone. This updating scheme makes use of the idea of a global moment of time: each cell must wait until a certain time before it can assume that all cells have finished the current step, and it is free to begin the next step.

Just as the notion of a global moment of time is inessential in physics, it is also inessential in cellular automata. An effectively synchronous updating can be achieved without it.

5.2.2 Local synchronization

Consider the example of a partitioning cellular automaton (see Section 2.3) with 2×2 block partitions. Since each block is updated as a unit, we don't need to keep a copy of the old state while constructing the new. We can perform a complete step of updating by processing each block of one partition separately; once all blocks have been updated, we switch to the other partition.[4]

Once four adjacent blocks on the first partition have been updated, the data for the block on the second partition that is made up of the four corners of these blocks is ready for the next step. Let us call the partition that is used during even numbered time steps the *even* partition, and the one used during odd numbered steps the *odd* partition. In general, if all four cells of a block on the even partition have been updated the same number of times, and this number is even, then the data in this block is ready to be updated again, and similarly for blocks on the odd partition.

Thus if we kept track of how many times each cell has been updated, we could forget about the global state of our automaton and simply follow the local rule: Update any block in which the data is ready to be updated. Depending on how often and in what order blocks are processed using this rule, we would get an enormous variety of global states for our automaton. However, any cell which is marked as having been updated n times will have exactly the same value it would have had if the whole automaton had been updated synchronously n times, and so once again we have an effectively synchronous evolution.

By only requiring that the data within a given block be ready, rather than the data for the whole space, we have eliminated the need for a global clock. If we call the number of times that a given cell has been

[4] This is exactly the strategy that CAM-7 (see Chapter 7) uses to perform an effectively synchronous updating without needing to buffer old cell values; it is also used in Section 6.4.1 in our discussion of Serial Quantum Compnters.

updated the *time* at that cell, then our locally synchronized updating scheme may result in configurations in which different parts of the CA space correspond to different moments of *time*. Because of causality, there are constraints on the possible patterns of times that we can find in our automaton. For example, two adjacent cells can never differ by more than a single unit of time, since they can only be updated *once* as parts of two different blocks before they must again be updated as parts of the same block. If we look at the pattern of times in our CA space, we may see hills and valleys, but no discontinuities.

5.2.3 Sychronization semaphores

Our scheme of the previous section involved blocks comparing cell times to see if they were equal, and also checking to see if they were odd or even. In fact, since adjacent cells can differ in time by no more than a single step, it is enough to compare whether adjacent cell times are both odd or both even, to tell if they are the same. Thus we need only store the *parity* of the time along with every cell, in order to make our local-synchronization scheme work.

To use an analogy, an ordinary synchronous cellular automaton is like a line of soldiers marching in step: all cells take a step forwards together. Our local synchronization scheme acts like a line of people walking forward hand-in-hand: no cell can can move forwards (i.e., be updated) until its neighbors have caught up. In this way they are basing their change on data from neighbors that have been updated the same number of times as they themselves have.

A piece of state information that is used to synchronize the operation of otherwise unsynchronized processes is called a *semaphore*. The general problem of making asynchronous logic simulate a synchronous cellular automaton using local synchronization semaphores is discussed in [80] and [71]; the somewhat simpler scheme based on *time parity* bits and partitioning that is described here will be sufficient for our purposes in this chapter.[5]

5.2.4 Asynchronous cellular automata

We will use the term Asynchronous Cellular Automata (ACA) to refer to systems in which the causal structure of a synchronous cellular automaton is simulated by an asynchronous system by means of local semaphores. One advantage of using such an *unclocked* logic scheme is that each cell is permitted to go as fast as it can: In a globally clocked scheme you must clock it more slowly than the time taken by the slowest *expected* cell, with some extra margin added for safety. A local scheme is held back only by the slowest *actual* cell.

In detail, the ACA synchronization scheme that we have developed abōve for the 2×2 partitioning automata might operate as follows: If the *time* for a given cell is *even*, then its next update (assuming we're

[5] If we add a time parity bit to each cell of a second-order automaton, a scheme that is analogous to the one we have discussed here allows local synchronization to replace a global clock.

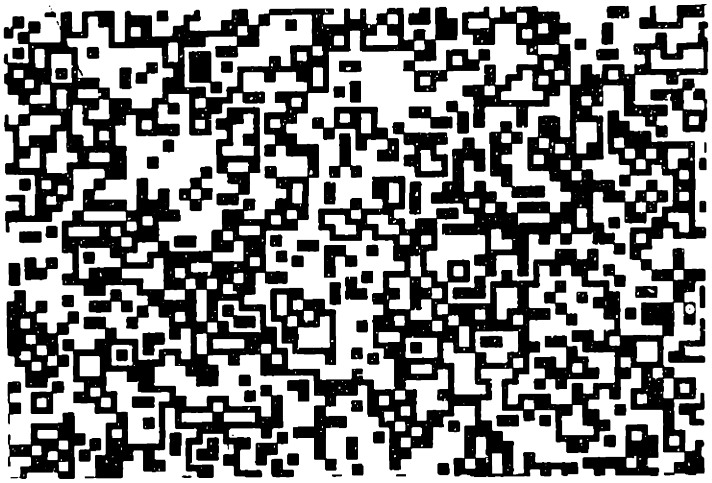


Figure 5.1: Time parity bits form a topographical map showing the isochrones.

only allowing forward steps, for the moment) must be as part of an *even* block. When all four cells of an even block have even times, then the updating should first make whatever changes to the cell values are mandated by the synchronous block rule we're simulating, and only then complement the time bits to record that these cells have been updated one additional time. Note that the cell data in all four cells doesn't have to change simultaneously; we only require that the new values must depend only on the state that held before the changes started. Once all cell data within the block has been updated, each cell's time-parity bit can be complemented. As soon as a cell's time-parity becomes *odd*, it is available to be updated as part of an *odd* block, using this same proceedure.

In Figure 5.1, we show a configuration of a two dimensional evolution that follows this scheme. Black indicates a time-parity of 1, white indicates 0. Connected regions of a single color all have values corresponding to the same moment of synchronous time. Thus the time parity bits trace out a sort of topographical map, in which *isochrones* take the place of level areas.

The relative time between any two cells can be reconstructed by following any path between the two cells. If we call an *odd row* one that passes through the upper left corner of an odd block, and similarly for odd columns, then whenever we cross from an even row to an odd row in phase with a change from an even time parity to an odd parity, then this is a step *backwards* in time. A step that is out of phase with the positional parity change is a step *forwards* in time.

In reversible partitioning schemes, such as the BBMCA model, we can imagine performing not only updating steps which take a block forwards, but backwards steps as well. As long as we only take steps whenever the cell data is ready for that step (either forwards or backwards) and change the time parity whenever we take a step, the evolution will never lose track of the correct relative times at all cells.[6]

Notice that the interpretation of our synchronization semaphores as time parity bits depends upon starting our system with a configuration of these bits that is compatible with a synchronous evolution; we will find that other possibilities are sometimes useful. For this reason, we will adopt the more neutral term of *guard bits* to refer to this syn-

[6] If we contemplate a reversible physical implementation of our RCA system, we would have to worry about implementing the synchronization scheme reversibly as well. A *thermodynamically* reversible version should certainly cause no difficulties (cf. [8]).

chronization information—these bits protect the causal structure of our computation from disruption. Synchronization based on guard bits will be used in Section 6.4.2, when we discuss a parallel model of quantum computation, as well as in the next section.

5.3 Relativity

In the previous section, we discussed ways of simulating the causal structure of a synchronous cellular automaton without recourse to non-local mechanisms. This led us to consider models which have different simultaneity properties than the synchronous systems that we began with. In this section we will discuss the relationship between these ACA models and Lorentz transformations.

5.3.1 Simultaneity

In Figure 5.2 we exhibit the causal structure of a 1-dimensional partitioning cellular automaton.[7] As usual, time increases going up the page, while space is spread across the page. The nodes correspond to the transition rule of this automaton; the arcs correspond to the cell values. A horizontal line (such as the dotted line marked *a*) through a set of arcs corresponds to the set of cell values that would be seen at a particular moment of synchronous time.

Any line which passes only through arcs and which never has a slope

[7] A similar diagram appears in [71].

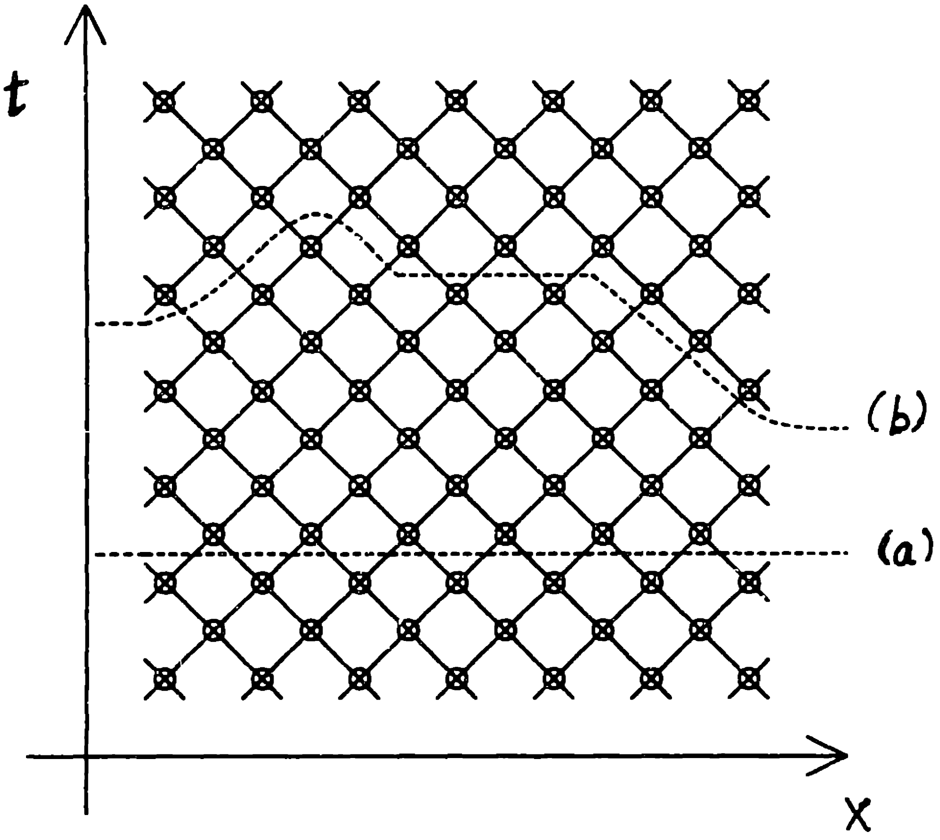


Figure 5.2: Spacetime diagram showing the causal structure of a one dimensional partitioning cellular automaton.

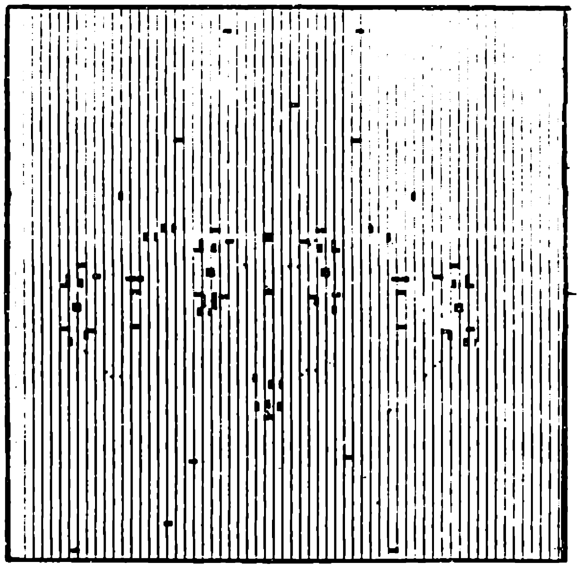


Figure 5.3: The circuit of Figure 2.8, updated on a different spacelike surface.

steeper than 45° (such as the dotted line marked b) represents a spacelike cut through our CA spacetime. Since all of these cuts have exactly the same number of cells on them, and since no cell value is ever more than one step ahead or behind its neighbor, we can actually represent the configuration represented by this cut in a cellular automaton using *guard* bits, as discussed in the preceding section.

Since the guard-bit synchronization scheme works in general, it works in particular in conjunction with a synchronous updating—we can add guard bits to our synchronous simulations of 2×2 block partitioning automata, and watch evolutions run on various spacelike surfaces. In Figure 5.3 we show a version of the BBMCA evolution of Figure 2.8, where the cells at the right edge correspond to times that are 128 steps later than those on the left. Because of periodic boundary conditions, this configuration has the property that points at the right, which correspond to the most advanced moment of time, are adjacent to points on the left, which correspond to the most retarded moment. There is, however, no discontinuity, since the guard-bits only record the fact that the slope (in time) is always positive to the right. The rule for the updating is that every block is updated during the first step in which its guard bits permit updating.[8]

In Figure 5.4, we show the same system, with the lower right corner 128 steps later than the upper left. In both figures, the particles travel more slowly going "downhill" than "uphill," since the fact that a particle is seen earlier as it moves "downhill" partially undoes its forward progress.

5.3.2 A Lorentz boost

The situations illustrated in these figures are closely related to Lorentz transformations—for illustrative purposes we will go through the correspondence in detail for the second figure, with the 45° "boost" (the

[8] Since each block of guard bits is either complemented or not, the parity of blocks on the unused partitions is conserved (see Section 3.1). Since all such guard-bit blocks would have zero parity in a normal synchronous updating, we can use the parity on these inactive blocks to tell us whether or not the evolution we're watching is equivalent to a synchronous updating, at least locally. A topologically preserved uniform slope, such as is present in Figure 5.3, is counted as synchronous by this accounting.

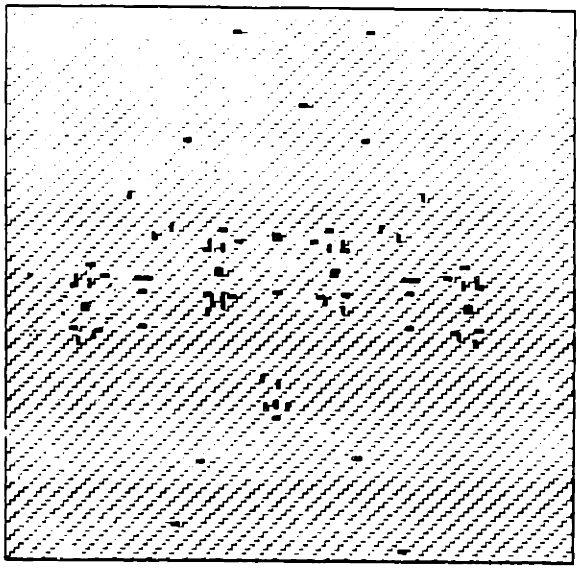


Figure 5.4: Same as previous figure, but with "boost" at a 45° angle.

rule supports particles that move at the maximum velocity—the "light speed"—parallel and perpendicular to this boost, and so the discussion is simpler). In order to understand what a normal Lorentz transformation might look like for these systems, it may help to imagine that the original system has somehow been physically built in such a way that the particles are actually realized by photons.[9]

Let us call the orginal system S, and the system in which a different simultaneity is being simulated S^*. In S^* we see as simultaneous events that were separated in S by an amount of time that increases linearly with distance. We can calculate the relative velocity that two inertial

[9] We could, for example, use short flashes of light for particles, and have appropriate photomultipliers, etc., inside all of the stationary "mirrors."

frames would have to have in order to produce the observed change in simultaneity.

In the original system S, the maximum speed that any particle could travel at was one diagonal position per time step—individual particles do this. For the purposes of this discussion, we will call such particles *photons*, and take their velocity, the *speed of light*, as our unit of velocity. Now consider two events that are simultaneous in S^*, and occuring in the upper left and lower right corners of Figure 5.4. The distance Δx in the original system S between these two events is 256 (in units of "light-travel-time"). The time difference Δt between these two events, for the synchronous evolution S, is 128 steps. In S^*, the two corners are simultaneous, as they would be in any frame that shares the simultaneity properties of S^*; therefore $\Delta t' = 0$ in our proposed moving frame, and

$$\Delta t' = 0 = \gamma(\Delta t - v\Delta x) \tag{5.1}$$

and so $v = \Delta t/\Delta x = \frac{1}{2}$. Thus this is a boost of 1/2 the speed of light. We will let S' denote S as seen from a "rocket" frame moving at this speed.

If we watch the particles in S^*, we find that photons travel at a speed of 1/2 perpendicular to the boost, at speed 1/3 "downhill," and at speed 1 "uphill."[10] This anisotropy arises because we have changed

[10] Since the diagonal lines, which represent steps in time, move at the speed of light down and to the right, and since a photon must lose two steps of forward travel each time it crosses one of these lines, these speeds are easy to predict from the change in simultaneity we are imposing.

the definition of simultaneity without adjusting distances or giving the system a net drift velocity. Furthermore, there is an overhead in performing the synchronization simulation which has nothing to do with relativity, and this must be factored out.

As we have noted, events that are happening simultaneously in S^* at two opposite corners will have $\Delta x = 256$ and $\Delta t = 128$ in the original system S. Noting that $\Delta t = \frac{1}{2}\Delta x$, we can calculate what $\Delta x'$ (the distance between these two events in S') should be:

$$\Delta x' = \gamma(\Delta x - v\Delta t) = \gamma(\Delta x - \frac{1}{2}(\frac{1}{2}\Delta x)) = \frac{3}{4}\gamma\Delta x$$

Since the distance Δx^* we would measure in S^* in this direction would be just Δx, we must use

$$\Delta x' = \frac{3}{4}\gamma\Delta x^* \tag{5.2}$$

Similarly, if we made a mirror-clock such as the one depicted in Figure 5.5, in the original system S the time taken for one round-trip by the photon might be some time Δt. In the transformed system S^*, since photons only travel at speed 1/2 in this direction, the measured round-trip time $\Delta t^* = 2\Delta t$. Relativistically we would expect $\Delta t' = \gamma\Delta t$, and so we must use

$$\Delta t' = \frac{1}{2}\gamma\Delta t^* \tag{5.3}$$

Finally, in the "rocket" system S' we would expect to see the parts of our original system that were stationary drifting past at a velocity v up and to the left. Since the distances and times in S^* must be adjusted

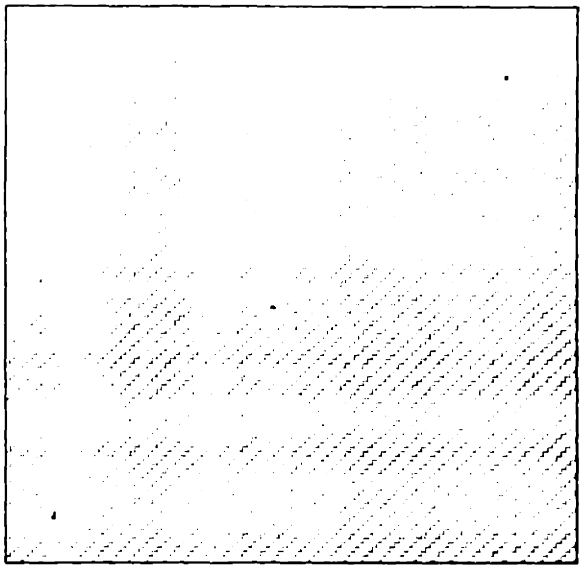


Figure 5.5: Two mirrors with a photon bouncing between them.

to get what we would see in S', the drift velocity that we must add in S^* to complete the correspondence is not simply v. If an object that is stationary in S is observed in S' to drift a distance $\Delta x'$ in a time $\Delta t'$, then the relative velocity seen in S' will be

$$v = \frac{\Delta x'}{\Delta t'} = \frac{\frac{3}{4}\gamma \Delta x^*}{\frac{1}{2}\gamma \Delta t^*} = \frac{3}{2}v^*$$

Thus for $v = \frac{1}{2}$, we must add a drift of $v^* = \frac{1}{3}$ up and to the left to our system S^*. If we imagine such a drift, and use equations (5.2) and (5.3) to convert from S^* distances to S' distances, then we can interpret S^* as being a Lorentz boosted system (for example, with this correction, photons will travel at unit speed in all four directions). Of course this drift could have been calculated directly from the observed speeds of photons in S^*, and the requirement that photons travel at the same speed "uphill" and "downhill."

Thus if we just watch the evolution of our transformed system S^*, we see qualitatively what we would in a true Lorentz transformation. The system will be stretched somewhat along the direction of the boost, but otherwise all of the instantaneous features of each configuration will be correct.

5.3.3 Lorentz invariance

Lorentz transformations are important in physics because Lorentz invariance seems to be a property of physical law. For the BBMCA system that we discussed above, there doesn't seem to be any important sense in which the law is form-invariant under a Lorentz transformation. Let us consider other candidates.

In RCA gas models such as those discussed in Section 4.2, there is only one particle velocity—everything moves with this one velocity, and all collisions conserve momentum. We could imagine classical particles that move at the speed of light, and which we could, in a gedanken experiment, start off on a grid so that at integer moments, all particles are found back on the grid, much as we did for the Billiard Ball Model of Section 1.3.

Now it is known that some of these lattice gas-systems, in the limit of slow flows, reproduce the isotropic Navier Stokes equation in terms of the behavior of particle densities[26]. It has been suggested[11] that such

[11] Tommaso Toffoli is the first to have suggested this, to my knowledge.

systems may actually be, in some statistical sense, Lorentz invariant. We have not reached a conclusion in this matter.

Chapter 6

Quantum Computation

When we describe the operation of a computer, we are of course describing the dynamical evolution of a physical system. What distinguishes a computer from other physical systems is its ability to simulate many aspects of other physical processes (including, in particular, the logical operation of any other computer, given enough time and memory[52]). As we have seen, for several recent models of computation the mapping between the computer and the underlying physics is quite direct. This has lead us to ask the question: How similar can the models used to describe computers be made to microscopic physics? This has an important bearing on the ultimate limitations of computing mechanisms, as we have seen.

We have presented a number of deterministic models of computation, including Fredkin's classical mechanical Billiard Ball model. But of course the world is quantum-mechanical, and so what we would really like is to understand the computational capabilities of microscopic quantum systems.

It may well be that to take best advantage of the computational capabilities of such systems we must reformulate our notion of a computation. However, in this chapter I will restrict my attention to the more straightforward problem of asking whether or not an ordinary deterministic computation can be described within the quantum formalism by a plausible Hamiltonian.[1] This discussion will be an extension of work by Benioff[4,5] who first raised this issue, and Feynman[23] who first exhibited a plausible Hamiltonian which could compute. I will use the BBMCA model of Section 2.4 as the basis of simple, uniform and explicit Hamiltonian models that embody the ideas of Benioff and Feynman, and then address for the first time the problem of constructing "quantum" models of parallel computation[47]. The synchroniza-

[1] We will not consider here the very interesting issue of a computer which is a Universal Quantum Simulator [22]. Such a computer would be a QM system which, started from an appropriate initial state corresponding to a state of any given QM system, would evolve in time t proportional to that taken by the given system into a QM state corresponding to the t-evolved state of the given system. Measurements performed on the simulator would correctly reproduce the QM statistics one would have obtained by performing an experiment on the original system. Such a simulator would provide an alternative to the present computational methods used to predict the consequences of QM models. Deutsch[18] discusses this problem, but doesn't address the important issue of the spatial locality of the Hamiltonian.

tion scheme discussed in Section 5.2 was invented specifically for this purpose, and the BBMCA makes it possible to consider simple and uniform models which only involve the interaction of nearby neighboring subsystems.[2]

6.1 Approaches to Quantum Computation

I will discuss two approaches that have been taken to the question of the possibility, in principle, of Quantum Computation (QC). Since the time-evolution operator in QM is always a unitary (and hence invertible) operator, both approaches are based on the notion of a reversible computer.[3] The two approaches are distinguished by whether the time-evolution operator or the Hamiltonian operator is taken as the starting point for the discussion.

6.1.1 Time-evolution operator approach

The first discussion indicating that QC was not necessarily inconsistent with the formalism of QM was that of Paul Benioff [4,5]. It depends upon the observation that the Schrödinger evolution of the wave func-

[2] A version of the material in this chapter appears in my paper *Quantum Computation*[47].

[3] The "irreversibility" of the measurement process will not bother us here, since we will be concerned with a dynamical evolution governed by the Schrödinger equation—there will be no measurements while the computation is proceeding.

tion is perfectly deterministic. If one associates a basis vector with each possible logical state of a reversible computer, then the one-step time-evolution which carries each state into the appropriate next state is a permutation on the set of basis states, and so is given by a unitary operator. Formally, it is always possible to write down an hermitian operator whose complex exponential equals this unitary operator. Given an initial logical-basis state, the Schrödinger evolution generated by this Hamiltonian will give the appropriate successor logical states at consecutive integer times.[4]

For example, if we let the possible configurations of the three state "computer" described in Figure 6.1 be represented by

$$A = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \qquad B = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \qquad C = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

then the time evolution given in Figure 6.1 can be represented by the unitary single-time-step operator

$$U = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

[4] Although the Schrödinger equation is a linear differential equation, in QM we allow a large enough set of basis vectors (one per configuration) so that a unitary operator can take a computer through an arbitrary invertible sequence of configurations. In particular, there is no difficulty in having the computer compute such "non-linear" functions as logical AND and OR.

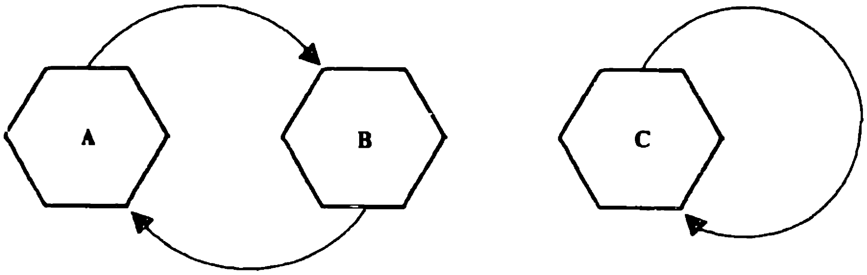


Figure 6.1: A simple 3-state machine. If the "computer" is in state A, it will go into state B. State B goes into A, and C doesn't change.

and from U we can find an hermitian matrix such that $U = e^{-iH}$. In this case,

$$H = \begin{pmatrix} \frac{\pi}{2} & -\frac{\pi}{2} & 0 \\ -\frac{\pi}{2} & \frac{\pi}{2} & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

6.1.2 Hamiltonian operator approach

One would like the Hamiltonian operator H to be given as a sum of pieces, each of which only involves the interaction of a few parts of the computer which are near to each other. The most direct way of ensuring that H is of this form is to write H down ab initio, rather than derive it from U.

Richard Feynman was the first to discuss this approach[23,6]. He realized that if the unitary operator F which describes one step of the desired forward evolution can be written as a *sum* of local pieces, then if we let $H = F + F^{\dagger}$ be the Hamiltonian operator, H will also be a sum of local (i.e., nearby-neighbor) interactions. The time-evolution operator $U(t) = e^{-iHt}$ is then a sum of powers of F and $F^{\dagger}$, taken with various weights. Thus if $|n\rangle$ corresponds to the logical state of a computer at step n (i.e., $F|n\rangle = |n+1\rangle$) then $U(t)|n\rangle$ is a superposition of configurations of the computer at various steps in the original computation. This superposition contains no configurations which aren't legitimate logical successors or predecessors to $|n\rangle$: if you make a measurement of the configuration of the computer, you will find it at some step of the desired computation. If instead you simply measure some piece of the configuration which tells you whether the computation is done or not, then when you see that it is done, you can immediately look elsewhere in the configuration to find the answer, and be assured that it is correct. Alternatively, one may construct a superposition of configuration states that acts as a sort of wave-packet state in which the computation moves forward at a uniform rate.

In order to write $F = \sum F_i$ with F a unitary operator, Feynman described a computer in which only one spot is active at a time. If instead of taking $\sum F_i$ to be unitary we only require the F_i's to be local, it turns out that we can describe a computer where all sites are active at once, but there is no longer a global time—synchronization becomes a matter of local intercommunication.

6.2 A reversible model of computation

We will illustrate the two approaches in terms of the BBMCA model of Section 2.4. This model is an obvious candidate for us to try to describe in terms of a lattice of QM spins. Here QM may even be superior to classical mechanics, since it is more natural to have identical two-state systems in QM (cf. [95,41]). In such an RCA model, during one logical step information has only to be communicated to nearby neighboring spins—data-paths are very short and so the impact of the finite light-speed restriction on computation speed is minimized.[5]

6.3 Time-evolution operator approach

In order to implement the BBMCA rule as a QM model, we will consider a two-dimensional lattice of spins, each of which is in a spin-component eigenstate with respect to the z-direction, which is taken to be perpendicular to the plane of the lattice. At each site, spin-up represents a logical 1, and spin-down a logical 0.

At a given lattice site with coordinates (i,j), the projection operator $P_{i,j} = (1+\sigma^z_{i,j})/2$ projects states which have a logical 1 at site (i,j), and the operator $\overline{P}_{i,j} = (1-\sigma^z_{i,j})/2 = 1 - P_{i,j}$ projects states with 0 at (i,j). The operator $a_{i,j} = (\sigma^x_{i,j} - i\sigma^y_{i,j})/2$ lowers a 1 at (i,j) to a 0, while $a^\dagger_{i,j} = (\sigma^x_{i,j} + i\sigma^y_{i,j})/2$ raises a 0 at (i,j) to a 1.

[5] For computations which can take advantage of this architecture. See Chapters 4 and 7.

We can now construct a unitary operator which will implement the BBMCA rule (Table 2.2) applied to a block of four sites, with upper-left-corner at position (i,j):

$$\begin{aligned} A_{i,j} \;=\; & (a_{i,j}a^{\dagger}_{i+1,j+1} + a^{\dagger}_{i,j}a_{i+1,j+1})\overline{P}_{i+1,j}\overline{P}_{i,j+1} \\ & +(a_{i+1,j}a^{\dagger}_{i,j+1} + a^{\dagger}_{i+1,j}a_{i,j+1})\overline{P}_{i,j}\overline{P}_{i+1,j+1} \\ & +(a_{i,j}a^{\dagger}_{i+1,j}a^{\dagger}_{i,j+1}a_{i+1,j+1} + a^{\dagger}_{i,j}a_{i+1,j}a_{i,j+1}a^{\dagger}_{i+1,j+1}) \\ & +1 - (\overline{P}_{i,j}\overline{P}_{i+1,j+1} + \overline{P}_{i+1,j}\overline{P}_{i,j+1} - 2\overline{P}_{i,j}\overline{P}_{i+1,j}\overline{P}_{i,j+1}\overline{P}_{i+1,j+1}) \end{aligned}$$

If $A_{i,j}$ is applied to a configuration of 1's and 0's, all of the lattice sites except those in the block at (i,j) will remain unchanged—this block will change according to the BBMCA rule. If we let

$$U_0 = \prod_{i,j\,\mathrm{even}} A_{i,j} \quad , \quad U_1 = \prod_{i,j\,\mathrm{odd}} A_{i,j}$$

then $U = U_1U_0$ is a unitary operator which exactly implements the BBMCA rule.[6] $U(t) = U^{t/2}$ (t an even integer) will exactly correspond to a BBMCA evolution at even integral times.

Now we will try to write $U(t) = e^{-iHt}$, with H a sum of local pieces. We begin by noting that $A^2_{i,j} = 1$ (follows from the BBMCA rule). Therefore $((1 - A_{i,j})/2)^2 = (1 - A_{i,j})/2$, and $\exp(-i\frac{\pi}{2}(1 - A_{i,j})) = A_{i,j}$ (expand the exponential). If we let $H_{i,j} = \frac{\pi}{2}(1 - A_{i,j})$, then

$$U_0 = \prod_{ij\,\mathrm{even}} A_{i,j} = e^{-i\sum_{i,j\,\mathrm{even}} H_{i,j}} \quad , \quad U_1 = e^{-i\sum_{ij\,\mathrm{odd}} H_{i,j}}$$

[6] It has been suggested[4,56] that in order to construct a time independent H for a U such as this, it is necessary to know explicitly the configuration of 1's and 0's at each step of every possible computation in advance.

and $U(t) = e^{-iHt}$, where $H = \sum_{ij\,\mathrm{even}} H_{i,j}$ when the integer part of t is even, and $H = \sum_{ij\,\mathrm{odd}} H_{i,j}$ at odd times.

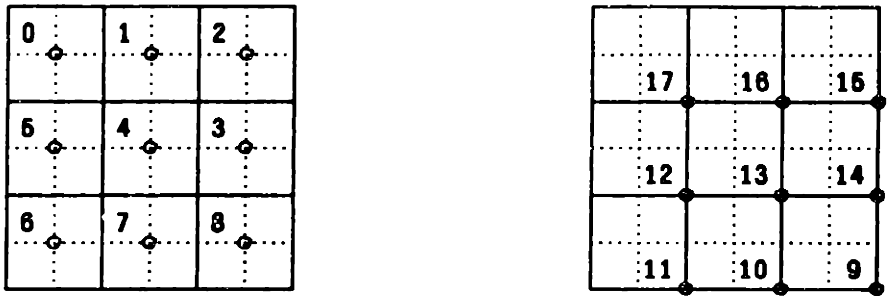
This U will reproduce the BBMCA evolution at all integer times. Intuitively, the reason we had to introduce a time dependence into H is because the $H_{i,j}$'s at a single time step all refer to non-overlapping blocks of spins, and so they all commute, allowing the product U_0 or U_1 of exponentials to be turned into an exponential of a sum. The even-block and odd-block $H_{i,j}$'s don't all commute—since the blocks overlap it makes a difference in which order the $H_{i,j}$'s are applied.

6.4 Hamiltonian operator approach

6.4.1 Serial computer

We can use Feynman's method to arrive at a time-independent version of the BBMCA.

We will use a 6×6 lattice (Figure 6.2) to illustrate the technique. The boundaries are periodic—we can imagine the lattice as being physically wrapped around into a torus, so that opposite edges touch. Now we divide a complete updating of the lattice into 18 independent steps, as shown in Figure 6.2. The step during which each 2×2 block is updated is indicated near its center, and (i_k, j_k) are the coordinates of the upper-left-hand corner of the $k^{\underline{th}}$ block. We introduce an extra "clock" spin at the center of each block, and let $c_k = \sigma_k^x - i\sigma_k^y$ be the lowering operator acting on this clock spin.



First 9 steps.

Next 9 steps.

Figure 6.2: A 6×6 lattice with periodic boundaries. All 2×2 blocks in the solid partition are updated, and then all blocks in the dotted partition. Because of periodicity, 9 through 17 mark the centers of dotted blocks.

We can now write the unitary operator F which in 18 steps accomplishes one complete updating of all the even and then all of the odd blocks on the lattice, as a sum of operators which each act on one block only:

$$F = \sum_{k=0}^{17} F_k \quad , \quad \text{where} \quad F_k = A_{i_k,j_k}\, c^{\dagger}_{k+1}\, c_k$$

and we start the lattice off with the clock-spin in block #0 up, and all of the rest of the clock-spins down.

If $|0\rangle$ is the initial state, then $F\,|0\rangle = |1\rangle$, the state where block #0 has been updated, and block #1 is waiting to be updated, $F\,|1\rangle = |2\rangle, \ldots, F\,|17\rangle = |18\rangle$, the state where one complete updating of all the blocks has been accomplished and the "up" clock-spin is in block #0, etc.

We have thus been able to write the forward time-step operator as a sum of local pieces by serializing the computation—only one block of the automaton is active during any given step.

Now we may write down a Hamiltonian operator $H = F + F^\dagger = \sum_k H_k$ (where $H_k = F_k + F_k^\dagger$) which is a sum of local interactions. If $|n\rangle$ is evolved for a time t, it becomes $e^{-iHt}\,|n\rangle$ which is a superposition of configurations of the serialized automaton which are legitimate successors and predecessors of $|n\rangle$.

We would like to make our automaton evolve forwards at a uniform rate—we can do this by constructing a wave-packet state. If we let N be the step-number[7] operator ($N\,|n\rangle = n\,|n\rangle$) then

$$\frac{d}{dt}\langle N\rangle = \left\langle\frac{[N,H]}{i}\right\rangle = \langle V\rangle \qquad \text{where}$$

$$V = \frac{[N,H]}{i} = \frac{F - F^\dagger}{i} \qquad , \quad [V,H] = 0$$

Thus the eigenstates of V have $\langle N\rangle$ which changes uniformly with time, and they can be chosen to be simultaneous eigenstates of H also. This allows us to make a superposition state from V's eigenstates which

[7] $|0\rangle$ is distinguished from $|\bar{1}8\rangle$ by looking at the computation part (as opposed to the clock spins part) of the state. To define N, we must choose some configuration of $\bar{1}$'s and 0's on each dynamical orbit of the BBMCA rule and call it the first configuration. For our "quantum" system we will only make use of states that have a single clock-spin up: step number zero corresponds to the up clock-spin being in block #0 of the first configuration.

has a fairly sharply-peaked step-number, and for which the computation proceeds at a uniform rate.

This corresponds closely to Feynman's original construction. Peres [56] noticed that we have the freedom to introduce coefficients ω_k multiplying each H_k, and that with an appropriate choice (neglecting for a moment the $A_{i,j}$'s) H becomes essentially the angular momentum operator J_x. This technique would allow us to start the system in state $|0\rangle$ and be assured of finding the system in state $|17\rangle$ after some prescribed time T that sets the scale for the ω_k's. However, the system would then undo its evolution, and be back in state $|0\rangle$ at time $2T$. Thus this technique is not useful for making our system run through a repeating computation cycle. If we want V to commute with H, then we are forced to set the ω_k's to a constant, as Feynman did.

This seems to be the best we can do with a serial computer that runs in a cycle. A Hamiltonian with a clock which gives exactly F when exponentiated (which is what we would ideally want) is necessarily nonlocal[57].

6.4.2 Parallel computer

In order to be able to write $F = \sum F_{i,j}$ with F a unitary operator, we described a computer in which only one spot was active at a time. We will now drop the restriction that $\sum F_{i,j}$ be unitary.

Let $H = \sum F_{i,j} + F_{i,j}^\dagger$. $U(t) = e^{-iHt}$ will now be a sum of terms involving all possible combinations of powers of the various $F_{i,j}$'s and

$F^\dagger_{i,j}$'s. If $U(t)\,|0\rangle$ is to be a superposition of configurations which correspond to legitimate classical evolutions from $|0\rangle$, then states where part of the automaton has been updated, while other parts haven't, must be allowed. This sort of cellular automaton where there is no global clock (as there has been in all of our preceeding discussion) is an asynchronous cellular automaton—such systems have been discussed in Section 5.2.

For an ACA to simulate an ordinary (synchronous) CA it uses a little extra state information, to tell the places that get ahead to wait for their neighbors to catch up. For a block rule such as the BBMCA, the additional information consists of a single *guard* bit added to every cell. The rule applied to a block will be the same as before, except that (for a forward step) an even block can only be updated if all four guard bits are 0's, and odd blocks only if they are all one's. When a block is updated, its guard bits are all complemented. As we have seen, this is sufficient to ensure that all four values in a block will always correspond to the same (synchronous) moment of time when they are updated. The guard bits will record the hills and valleys in time (relative numbers of cell updates) as a sort of topographical map, drawn using only 1's and 0's.

If we imagine that the guard bits are spins in a lattice that sits directly below our original BBMCA lattice, and let $g_{i,j}$ be the lowering operator for a spin at the site (i,j) on the guard-bit lattice, then our forward-step operator for the site (i,j) is given by $F_{i,j} = A_{i,j}\, g^{\dagger}_{i,j}\, g^{\dagger}_{i+1,j}\, g^{\dagger}_{i,j+1}\, g^{\dagger}_{i+1,j+1}$ for (i,j) even, $F_{i,j} = A_{i,j}\, g_{i,j}\, g_{i+1,j}\, g_{i,j+1}\, g_{i+1,j+1}$ for (i,j) odd, and $F = \sum F_{i,j}$ acting on a given configuration will produce a superposition of configurations, each of which has advanced one step at some location.

$F^{\dagger}_{i,j}$ has the g's and $g^{\dagger}$'s interchanged, relative to the definition of $F_{i,j}$, and so it implements a possible step *backwards* rather than *forwards*.

For both even and odd blocks, $H_{i,j} = F_{i,j} + F^{\dagger}_{i,j} = A_{i,j}(\, g^{\dagger}_{i,j}\, g^{\dagger}_{i+1,j}\, g^{\dagger}_{i,j+1}\, g^{\dagger}_{i+1,j+1} + g_{i,j}\, g_{i+1,j}\, g_{i,j+1}\, g_{i+1,j+1})$, and $H = \sum_{\text{even or odd blocks}} H_{i,j}$

This model can be made to perform a computation by occasionally checking for a "computation done" flag—some particular group of cells which the computation will set to certain values when it is done. The appearance of such a flag ensures that there is an unbroken chain of sites that connect the flag to the place that signaled it to appear, none of which can correspond to moments of time in the equivalent synchronous evolution that precede the moment the signal passed that site. Thus if the flag signal was produced by a process that first put the answer somewhere, the answer must still be available there when the "done flag" is seen.

Of course what we would really like to do is to show that we can make this sort of computer run at a uniform rate. The difficulty here is that if we let N be an operator which, when applied to a configuration state, returns the average synchronous-step in that configuration, and

$V = [N, H]/i$, we find that V doesn't commute with H, and so the situation is more complicated than it was in the serial-computer case.

I don't know if this computer can be made to "run" in a reasonable fashion. Perhaps it would be worthwhile to study ways in which such a model could be driven by an external field—turning it into a thermodynamically reversible model, á la Bennett.

6.5 Discussion

As it is quantum mechanics which today embodies our most fundamental understanding of microscopic physical phenomena, in our quest for faster and more efficient computation we are naturally led to the problem of trying to describe computing mechanisms which operate in an essentially quantum-mechanical manner.

In this chapter I have attempted to extend Feynman's method for constructing a "quantum" Hamiltonian model of computation in order to arrive at a model in which the parallelism inherent in the operation of physical law simultaneously everywhere is put to use. The combination of parallelism and locality of interaction makes such models asynchronous: I introduced a local synchronization scheme to allow such a model to simulate a synchronous causality. I arrived at a model which, started from any state on a given (asynchronous) computational orbit, will only visit other states on this same orbit, or superpositions of such states. It is not clear how to make such a model run at a uniform rate.

Chapter 7

Cellular Automata Machines

The charter of the Information Mechanics Group at the MIT Laboratory for Computer Science has been to study the physical bases of computation, and the computational modeling of physics-like systems. This has led us to study areas such as reversible computation and cellular automata.

In 1981, our frustration with the capabilities of conventional computers for simulating cellular automata became acute when we tried to study the large-scale, long-term behavior of some remarkable reversible rules. We were also excited by the prospect that with more appropriate hardware one would be able to run cellular automata at the speed of a movie rather than that of a slide show, and actually watch their time-averaged macroscopic evolution.

Tom Toffoli built the first prototype at home, using a few TTL chips—a sequential machine that scanned a two-dimensional array of cells, producing new states for the cells fast enough and in the right order so that it could keep up with the beam of an ordinary raster-scan television monitor. After a few years of experimentation and refinement, we arranged for a version of our machine, namely CAM-6, to be produced commercially, so that others could get in on the fun[14,80]. The architecture of this machine is discussed in Section 7.3, and my design for a much larger machine is discussed in Section 7.6.

The existence of CAM's (Cellular Automata Machines) has already had a direct impact on the subject of CA simulations of fluid mechanics. In playing with gas-like models, we found one that Yves Pomeau had previously investigated—the HPP gas[31].[1] As Pomeau tells us, seeing his CA running on our machine made him realize that what had been conceived primarily as a *conceptual* model could indeed be turned, by using suitable hardware, into a *computationally accessible* model, and stimulated his interest in finding CA rules which would provide better models of fluids[26].

In fact (as we shall see below) the advantages of an architecture optimized for CA simulations are so great that, for sufficiently large experiments, it becomes absurd to use any other kind of computer.

[1] Pomeau's result was brought to our attention by Gérard Vichniac, then working with our group.

7.1 Truly massive computation

Cellular automata constitute a general paradigm for massively parallel computation. In CA, size and speed are decoupled—the speed of an individual cell is not constrained by the total size of the CAM. Maximum size of a CAM is limited not by any essential feature of the architecture, but by economic considerations alone. Cost goes up essentially linearly with the size of the machine, which is indefinitely extendible.

These properties of CAM's arise principally from two factors. First, in conventional computers, the cycle time of the machine is constrained by the finite propagation speed of light—the universal speed limit. The length of signal paths in the computer determines the minimum cycle time, and so there is a conflict between speed and size. In CA, cells only communicate with spatially adjacent neighbors, and so the length of signal paths is inherently independent of the number of cells in the machine. Size and speed are decoupled.

Second, this locality permits a modular architecture: there are no addressing or speed difficulties associated with simply adding on more cells. As you add cells, you also add processors. Whether your module of space contains a separate processor for each cell or time-shares a few processors over many cells is just a technological detail. What is essential is that adding more cells doesn't increase the time needed to update the entire space—since you always add associated processors at a commensurate rate. For the forseeable future, there are no practical technological limits on the maximum size of a simulation achievable with a fixed CAM architecture.

The reason that CA can be realized so efficiently in hardware can ultimately be traced back to the fact that they incorporate certain fundamental aspects of physical law, such as locality and parallelism. Thus the structure of these computations maps naturally onto physical implementations. It is of course exactly this same property of being physics-like that makes CA a natural tool for physical modeling (e.g., fluid behavior). Von Neumann-architecture machines emulate the way we consciously think: a single processor that pays attention to one thing at a time. CA emulate the way nature works: local operations happening everywhere at once. For certain physical simulations this latter approach seems very attractive.

7.2 A processor in every cell?

In order to maintain the advantages of locality and parallelism, CAM's should be constructed out of modules, each representing a "chunk" of space. The optimal ratio of processors to cells within each module is a compromise dictated by factors such as

- technological and economic constraints,
- the relative importance of speed versus simulation size,
- the complexity and variability of processing at each cell,
- the importance of three-dimensional simulations,
- I/O and inter-module communications needs, and

- a need for analysis capabilities of a less local nature than the updating itself.

Just to give an idea of one extreme at the fine-grained end of the spectrum, consider a machine having a separate processor for each cell, and some simple two-dimensional cellular-automaton rule built in.[2] We estimate that, with integrated-circuit technology, a machine consisting of 10^{12} cells and having an update cycle of 100 pico-seconds for the entire space will be technologically feasible within 10 years. If the same order of magnitude of hardware resources contemplated for this CAM (using the same technology) were assembled as a serial computer with a single processor, the machine might require seconds rather than picoseconds to complete a single updating of all the cells.

There are serious technological problems which must be overcome before three-dimensional machines of this maximally-parallel kind will be feasible. The immediate difficulty is that our present electronic technologies are essentially two-dimensional, and massive interconnection of planar arrays (or "sheets") of cells in a third dimension is difficult. In the short term, this problem can be addressed by time-sharing relatively few processors over rather large groups of cells on each sheet; this allows interconnections between sheets to also be time-shared. The architectures of the CAM's built by our group make use of this idea.

[2] This approach does not necessarily restrict one to a single specific application. There are simple universal rules (cf. LOGIC in Section 4.4) which can be used to simulate any other 2-dimensional rule in a local manner.

A more fundamental problem which will eventually limit the size of CAM's is heat dissipation: heat generation in a truly three-dimensional CAM will be proportional to the number of cells, and thus to the volume of the array, while heat removed must all pass through the surface of this volume. As we have discussed in Chapter 1, this problem can in principal be reduced dramatically by employing reversible logic, and a universal RCA rule such as the BBMCA rule of Section 2.4.

7.3 An existing CAM

CAM-6 is a cellular automata machine based on the idea that each space-module should have few processors and many cells. In addition to drastically reducing the number of wires needed for interconnecting modules (even in two dimensions) this allows a great deal of flexibility in each processor while still maintaining a good balance between hardware resources devoted to processing and those devoted to the storage of state-variables (i.e., cell states).

Each CAM-6 module contains 256K bits of cell-state information and eight 4K-bit lookup tables which are used as processors. Both cell-state memory and the processors are ordinary memory chips, similar to those found in any personal computer. The rest of the machine consists of a few dozen garden-variety TTL chips, and one other small memory chip used for buffering cell data as it is accessed. All of this fits on a card that plugs into a personal computer (we used an IBM-PC, because of its ubiquity) and gives a performance, in many interesting

CA experiments, comparable to that of a CRAY-1.[3]

The architecture which accomplishes this is very simple.

Cell-state memory is organized as 65536 cells in a 256×256 array, with 4 bits of state in each cell. The cell states are mapped as pixels on a CRT monitor. To achieve this effect, all 4 bits of a cell are retrieved in parallel (with the array being scanned sequentially in a left-to-right, top-to-bottom order). The timing of this scan is arranged to coincide with the framing format of a normal raster-scan color monitor—cell values are displayed as the electron beam scans across the CRT. Thus a complete display of the space occurs 60 times per second.

Such a memory-mapped display is very common in personal computers. What we add (see Figure 7.1) is the following: as the data streams out of the memory in a cyclic fashion, we do some buffering (with a pipeline that stretches over a little more than two scan lines) so that all the values in a 3×3 window (rather than a single cell at a time) are available simultaneously. We send the center cell of this window to the color monitor, to produce the displav as discussed above. Subsets of the 36 bits of data contained in this window (and certain other relevant signals) are applied to the address lines of lookup tables:

[3] For the simulation of extremely simple CA rules, without any simultaneous analysis or display processing, any computer equipped with raster-op hardware will be able to perform almost as fast as CAM-6, since this CAM is really just a specialized raster-op processor. These computers will not be able to compete as the processing becomes more sophisticated, or as we add more modules to simulate a bigger space without any slowdown.

the resulting 4 output bits are inserted back in memory as the new state of the center cell. In essence, the set of neighbor values is used as an index into a table, which contains the appropriate responses for each possible neighborhood case. Even when a new cell state has been computed, the above-mentioned buffering scheme preserves the cell's current state as long as it is needed as a neighbor of some other cell still to be updated, so that every 60-th of a second an updating of the entire space is completed exactly as if the transition function had been applied *to all cells in parallel.*

Four of the eight available lookup-table processors are used simultaneously within each module, each taking care of updating 64K bits

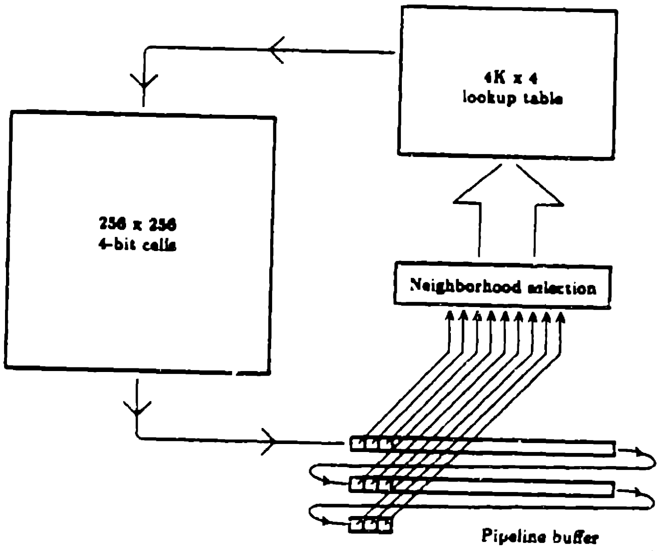


Figure 7.1: As the 4 planes are scanned, a stream of 4-bit cell values flow through a pipeline-buffer. From this buffer, 9 cell values at a time are available for use as neighbors. Of these 36 bits, up to 12 are sent to the lookup table, which produces a new 4-bit cell value.

of cell-state. The other four *auxiliary* lookup tables can be used, in conjunction with a color-map table and an event-counter, for on-the-fly data analysis and for display transformations. They can also be used directly in cell updating. A variety of neighborhoods are available, each corresponding to a particular set of neighbor bits and other useful signals that can be applied as inputs to the lookup tables. These neighborhoods are achieved by hardware-multiplexing the appropriate signals under software control of the personal-computer host.

Most of CAM-6's power derives from this use of fast RAM tables (which c*n accomplish a great deal in a single operation) as processors.

Connectors are provided to allow external transition-function hardware (such as larger lookup tables or combinational logic) to be substituted for that provided on the CAM-6 module. Such hardware only needs to compute a function of neighborhood values supplied by CAM-6, and settle on a result within 160 nanoseconds. The CAM-6 module takes care of ւpplying this function to the neighborhood of each cell in turn and storing the result in the appropriate place. If the external source for a new cell-value is a video camera (with appropriate synchronization and A/D conversion), then CAM-6 can be used for real-time video processing.

Th. connectors also allow external signals to be brought into the module as neighbors, allowing the output of an external random number generator, or signals from other CAM-6 modules, to be used as arguments to the transition function. When several modules are used together they all run in lockstep, updating corresponding cell positions simultaneously. Three-dimensional simulations can be achieved by having each module handle a two-dimensional slice, and *stacking* the slices by connecting neighbor signals between adjacent slices.

The hardware resources and usage of CAM-6 are discussed in more detail in the book *Cellular Automata Machines: a new environment for modeling*[80]. For illustrative purposes a few of the physical modeling examples discussed in this book will be surveyed in the next section (some of these are discussed in more detail in Chapter 4).

7.4 Physical modeling with CAM-6

CAM-6 (simply 'CAM' in this section) is a general-purpose cellular automata machine. It is intended as a laboratory for experimentation, a vehicle for communication of results, and a medium for real-time demonstration.

The experiments illustrated in this section were performed with a single CAM module, with no external hardware attached.

Time correlations. Figure 7.2 shows the results of some time-correlation experiments that made use of CAM's event counter. In these simulations, two copies of the same system were run simultaneously, each using half of the machine. Corresponding cells of the two systems were updated at the same moment. Each run was begun by initializing both systems with identical cell values, and then holding one of the

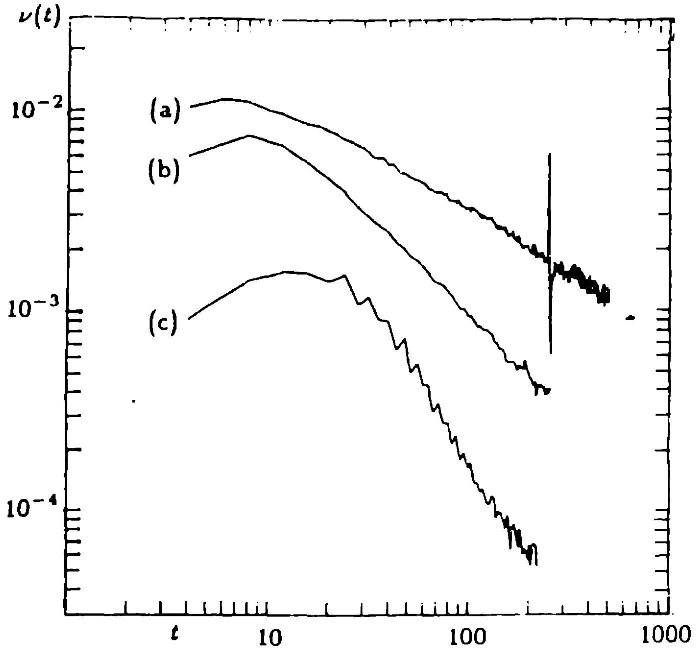


Figure 7.2: Time-correlation function $\nu(t)$ for **HPP-GAS** (a), **TM-GAS** (b), and **FHP-GAS** (c).

systems fixed while updating the other a few times. The systems were then updated in parallel for several thousand steps, with a constant time-delay between the two versions of the same system. Velocity-velocity autocorrelations were accumulated by comparing the values of corresponding cells as they were being updated, and sending the results of the comparisons to a counter that was read by the host computer between steps. In addition to time-correlations, space and space-time correlations could similarly be accumulated simply by introducing a spatial shift between the two systems before beginning to accumulate correlations. The three time-correlation plots refer to three different lattice gases; each data point represents the accumulation of over a billion comparisons. The whole experiment entailed accumulating about 3/4 of a trillion comparisons, and took about two-and-a-half days to run.

Self diffusion. Figure 7.3 is a histogram showing the probability that a particle of the **TM-GAS** lattice gas (see Section 4.2.1) started at the origin of coordinates will be found at a position (x, y) after some fixed number of steps (1024 steps in this case).[4] The data was accumulated by "marking" one of the particles (using a different cell value for it than for the rest, but not changing its dynamics) and then using the auxilliary lookup tables in combination with the event counter to track its collisions, and hence its movements. For each (x, y) value the height of the plot indicates the number of runs in which the particle ended up

[4] This experiment was conducted by Andrea Califano.

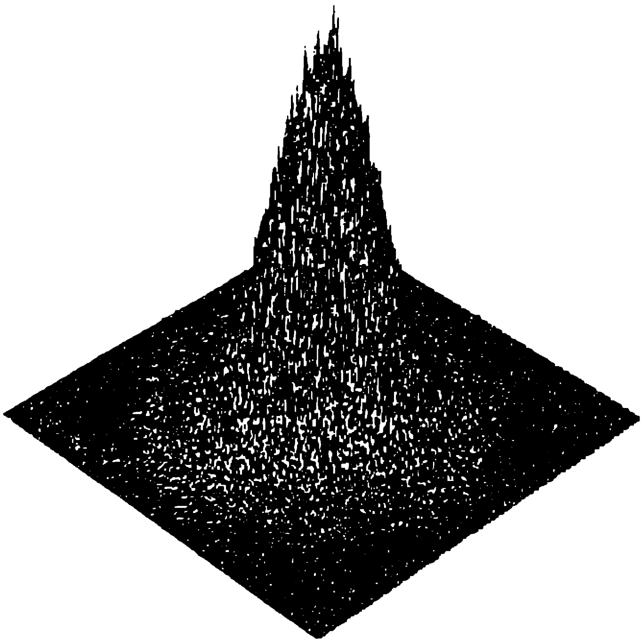


Figure 7.3: Histogram of $P(x, y; t)$—the probability that a particle of **TM-GAS** will be found at x, y at time t—as determined by a long series of simulation runs on CAM.

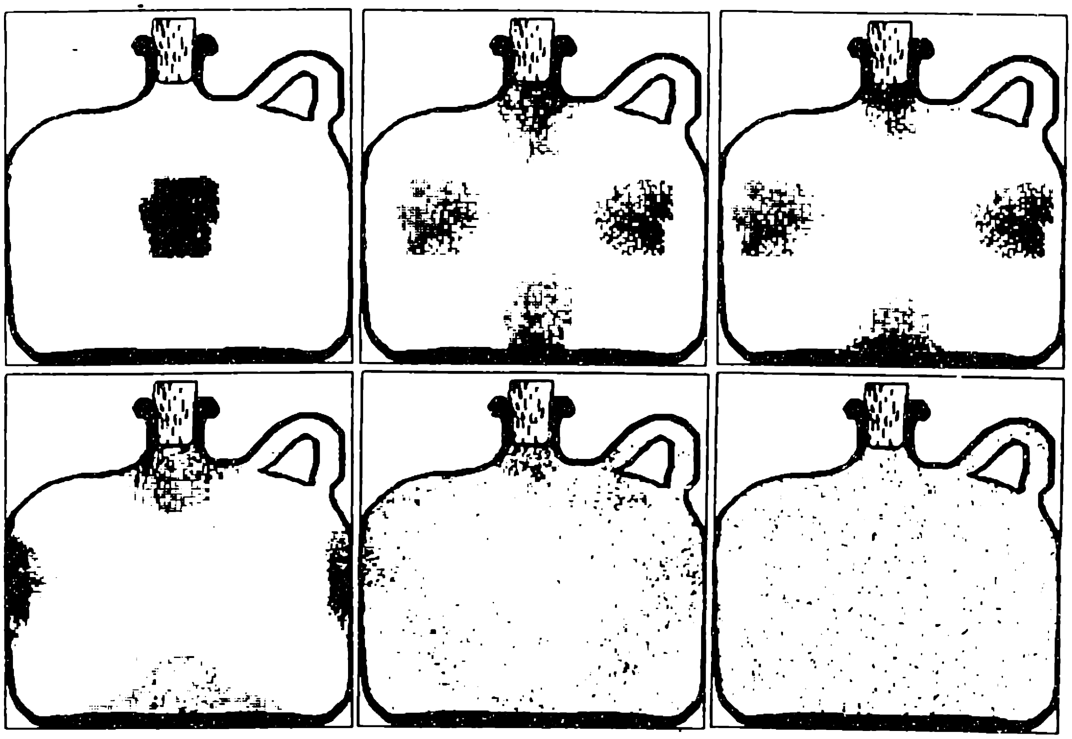


Figure 7.4: Expansion of a TM-GAS cloud in a vacuum. Repeated collisions between particles and with container's walls eventually lead to thorough thermalization.

at that point.

Though such an experiment requires a massive amount of computation, the essential results of each run can be saved in a condensed form (as a string of collision data for a single particle) for post-analysis. In this way, a single experiment can be used for studying various kinds of correlations.

Thermalization. Figure 7.4 shows the expansion of a clump of particles of TM-GAS. In this experiment one bit of state within each cell is devoted to indicating whether or not that cell contains a piece

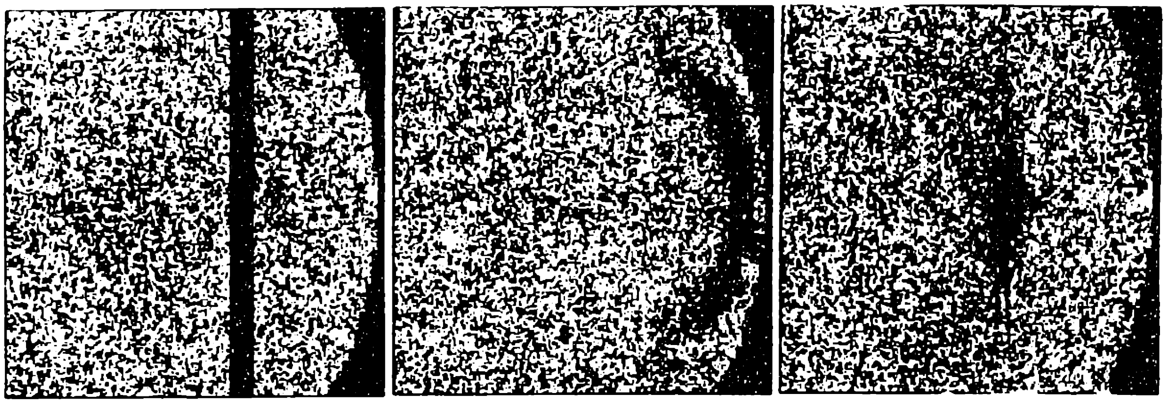


Figure 7.5: A plane pulse traveling towards a concave mirror (a) is shown right after the reflection (b) and approaching the focal point (c).

of the wall; this bit represents a boundary-condition *parameter* of the simulation, and doesn't change with time. Other state information in each cell is used to simulate the moving gas. Cells which don't border on a wall follow the TM-GAS rule. Near a wall, the rule is modified so that particles are reflected. An arbitrary boundary can be simulated simply by drawing it—here we've drawn a jug. Initially it is evident that there are only four directions of travel available to the particles, but as the gas equilibrates this microscopic detail becomes invisible.

Reflection and refraction. Figure 7.5 shows exactly the same kind of simulation as Figure 7.4, but with a different initial condition. Here we've drawn a wall shaped as a concave mirror, and illustrate reflection of a density enhancement which is initially travelling to the right. For compactness, we use here a special kind of high-density nondissipative wave (a "soliton") that this rule supports (on a slightly larger scale, such phenomena can of course be demonstrated with ordinary near-

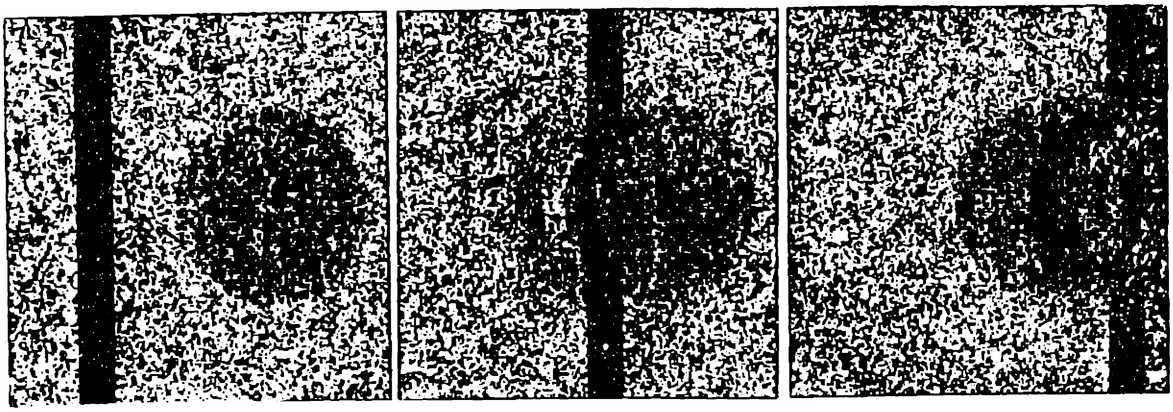


Figure 7.6: Refraction and reflection patterns produced by a spherical lens.

equilibrium "acoustic" waves).

In a similar experiment, Figure 7.6 shows the refraction of a wave by a lens. As before, we draw our obstacle by reserving one bit of each cell's state as a spatial parameter denoting whether the cell is inside or outside the lens. Particles outside the lens follow a lattice-gas rule. Inside the lens, this rule is modified so that particles travel only half as fast as outside (this is accomplished simply by having the particles move only during half of the steps). Rules that depend on time in such a manner are provided for in CAM's hardware by supplying "pseudo-neighbor" signals that can be seen simultaneously by every cell as part of its neighborhood, and can be changed between steps under software control.

- *Tracing a flow.* Figure 7.7 illustrates an experiment in which smoke is used to trace the flow of a lattice gas. Frame (a) shows a lattice gas with a net drift to the right—this is not evident if we don't color the particles to indicate their velocities. Frame (b) shows the diffusion

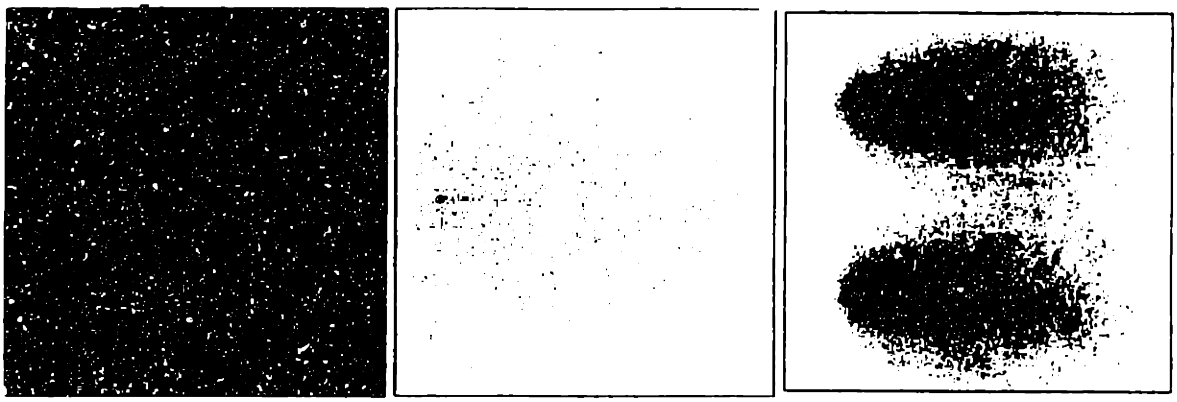


Figure 7.7: (a) The direction of drift is invisible if the fluid has uniform density. (b) Markers ejected by a smokestack diffuse in the fluid. (c) On a larger-scale simulation, the streamlines start becoming visible.

of particles released from a single point. This source is implemented in the same manner as the mirrors and lenses discussed previously—we mark the cells that are to be sources, and follow a different rule there. The "smoke" particles released from this source are colored differently from the other particles; however, the dynamics is "color-blind," and treats them just as ordinary gas particles. By looking only at these diffusing smoke particles, one can immediately see their collective net drift. Frame (c) shows the same phenomenon as (b), but using a space 16 times larger (1024×1024 rather than 256×256). Since the width of the diffusion pattern is proportional to $\sqrt{t}$, whereas the net distance a particle drifts is proportional to t, the drift becomes more and more evident as the scale is increased.

The larger cellular automaton shown in that last frame was simulated by a single CAM module,[5] using a technique called *scooping.* The

[5] This experiment was conducted by Tom Cloney.

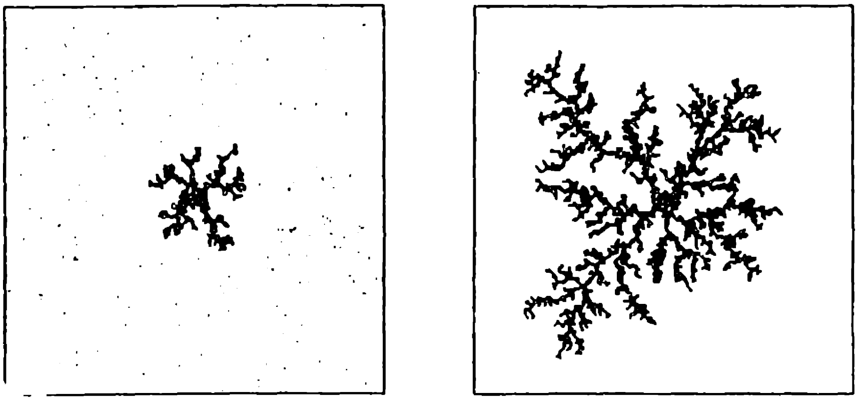


Figure 7.8: Dendritic growth by diffusion-limited aggregation. The process was started from a one-cell seed in the middle, and with a 10% density of diffusing particles.

1024×1024 array of cells resides in the host computer's memory, and CAM's internal 256×256 array is used as a cache: this is loaded with a portion of the larger array, updated for a couple of dozen steps, and then stored back; the process is repeated on the next portion, until all of the larger array has been updated. Since scooping entails some overhead (data must be transfered between main memory and cache, and data at the edges of the cache—where some of the neighbors are not visible—must be recomputed in a later scoop), the effective cell-update rate drops somewhat, but to no worse than about half of CAM's normal rate. A similar technique can be used for three-dimensional simulations with a single CAM (this works particularly well with partitioning rules—see Section 2.3).

Diffusion-limited aggregation. Figure 7.8 shows two stages in the growth of a dendritic structure by a process of diffusion-limited aggregation[66,88]. There are three coupled systems here, each using one bit of each cell's state. The first system is a lattice gas with a 50% density of particles. This gas is used only as a "thermal bath" to drive the diffusion of particles in a second system. The contents of the cells in this second system are randomly permuted in a local manner that depends on the thermal bath. The third system is a growing cluster started from a seed consisting of a single particle: whenever a particle of the diffusing system wanders next to a piece of the cluster, the particle is transferred to the cluster system, where it remains frozen in place. Owing to this capture process, there will be fewer diffusing particles near the growing cluster than away from it, and the net diffusion flow is directed toward the cluster. Most of the new arrivals get caught on the periphery of the cluster, giving rise to a dendritic pattern.

Ising spin systems. Figure 7.9 contains two views of a deterministic Ising dynamics[17,85,59,34]: both frames correspond to a single configuration of spins. The one on the left shows the spins themselves, the one on the right illustrates the use of CAM's auxiliary tables to display in real-time a *function* of the system's state rather than the *state* itself—in this case, the bond energy. One can watch the motion of this energy (which is a conserved quantity and thus obeys a continuity equation) while the evolution is taking place; one can run space-time correlation experiments on either magnetization or energy, etc. By us-

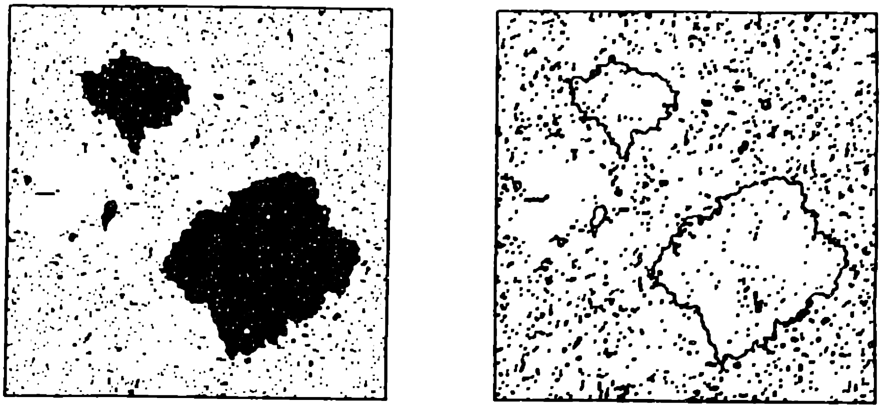


Figure 7.9: (a) A typical spin configuration; (b) the same configuration, but displaying the energy rather than the spins.

ing a heat bath (as in the preceding aggregation model) one can also implement canonical Ising models. Figure 7.10 plots the magnetization in such a model versus the Monte Carlo acceptance probability.[6] Techniques which allow CAM itself to generate (in real-time) the finely-tunable random numbers needed to implement the wide range of acceptance probabilities used in this experiment are discussed in [80]. The actual method used in the experiment plotted here involved using a second CAM machine for this purpose and taking advantage of an instant-shift hardware feature that happens to be present in CAM-6; this feature is central to the design of CAM-7.

Other phenomena. Other physical phenomena for which CAM-6 models are provided in [80] include nucleation, annealing, erosion, ge-

[6] This experiment was conducted by Charles Bennett.

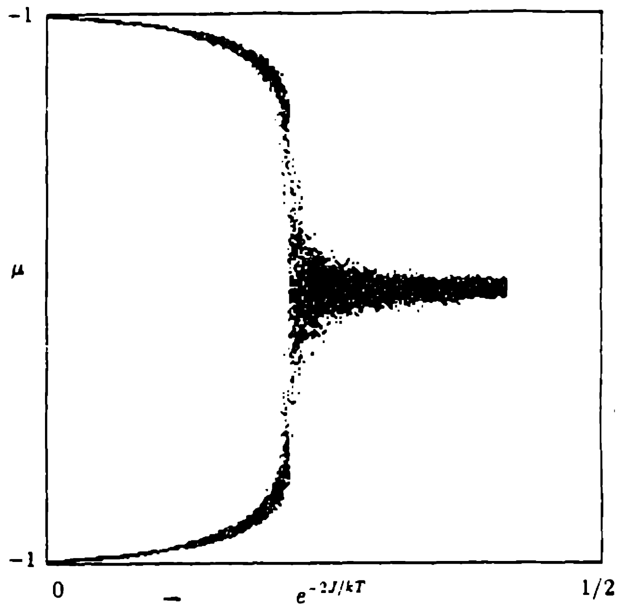


Figure 7.10: Magnetization μ in the canonical-ensemble model, versus the Monte Carlo acceptance probability. Note the sharp transition at the critical temperature T_{crit}.

netic drift, fractality, and spatial reactions analogous to the Zhabotinsky reaction. A number of models which are interesting for the study of the physics of computation are also given, including the BBMCA model of Section 2.4 and some models of asynchronous computation. The examples in our book were developed to illustrate a variety of techniques for using CAM-6; they may also serve to clarify what we mean when we call this device a general-purpose cellular automata machine.

7.5 CAM-7

If we scale CAM-6 up sixteen-thousandfold we arrive at a machine with hardware resources comparable to those of a large mainframe computer, but arranged in a manner suitable for extensive scientific investigations using cellular automata. In this and subsequent sections we will describe our plan for this CAM-7 machine; this design is still undergoing development.

The principal hardware specifications of CAM-7 will be:

- 2 gigabits of cell-state memory (120ns dynamic RAM)
- 1/2 gigabit of lookup-table memory (35ns static RAM)
- 8192 plane-modules (each 512×512) operating in parallel
- 200 billion cell-bit updates per second (8192 every 40ns)
- I/O bus 8192 bits wide, with a 40ns synchronous word rate (all data appears on this *flywheel bus* once each step)
- 2-dimensional simulations on a 16384×8192×16 region
- 3-dimensional simulations on a 512×512×512×16 region
- any 512×512 region can act as its own TV frame buffer
- any 16 bits in a 1025×1025 region can be used as a neighborhood

As few as 16 of the plane-modules that constitute a complete CAM-7 machine could be assembled into a 512×512×16 fractional machine capable of performing 400 million cell-bit updates per second. Such a machine could be integrated into a personal computer much as CAM-6 was, at a similar cost. As many as 100 or more complete CAM-7 machines could be connected together, to perform much larger two or three dimensional simulations—the constraints are really economic rather than technological.

7.6 CAM-7 Architecture

This machine's speed comes from its parallelism: the machine is made out of ordinary commodity RAM chips, driven at full memory bandwidth, plus some rather simple "glue" logic which will almost all go into a semi-custom controller-chip associated with each plane-module. We feel that this restriction to inexpensive memory is important, since it should make it economically feasible to build several CAM-7 machines and connect them together to perform CA experiments which involve many trillions of updates per second.

7.6.1 Basic structural elements

The design really consists of two separate parts: a "data flywheel" which sequentially runs through all the cell data once each step, and lookup tables which transform the cell data as it passes through them.

The data flywheel is made up of 8192 plane-modules, each of which is a $512\times512\times1$ array of bits. The scanning of a module proceeds as for a memory-mapped display (just as it did for CAM-6). Each module puts out one bit every 40 nanoseconds, and takes in one bit at the same time.

The lookup tables are each connected to 16 plane-module outputs. Every 40 nanoseconds they return a set of 16 new cell values which are injected back into the modules (see Figure 7.11).

This selection of module size and update rate is such that the

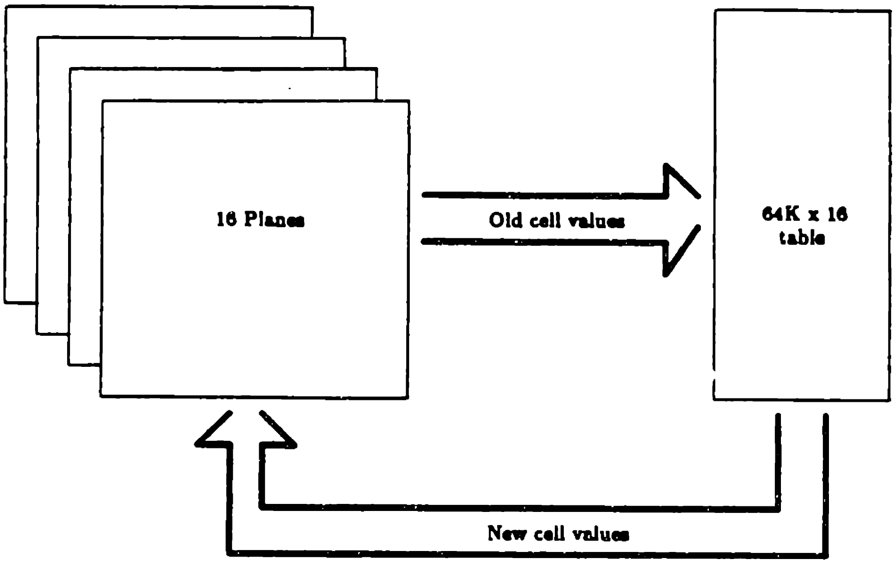


Figure 7.11: A layer of CAM-7, consisting of 16 plane-modules. As the planes are scanned, a stream of 16-bit cell values are sent as addresses to a 64K×16 lookup table—the 16-bit results are put back into the planes, as the new cell values.

scanning of the modules can be locked to the framing format of a high-resolution monitor, so as to display 512×512-pixel images at 60 frames/sec with no interlacing. When so locked, CAM-7 will update its two-gigabits of cell memory 60 times per second. If we decouple the updating from the TV frame rate, CAM-7 will be able to update this entire two-gigabits 100 times per second. When decoupled, we can have each plane module scan only a fraction of its cells, permitting many more updates per second of this smaller array. For example, if each module scans a region that is only 64×64, then CAM-7 will be able to update a space of size 2048×1024×16 about 4000 times per second.

7.6.2 Neighborhoods

The most significant architectural difference between CAM-6 and CAM-7 lies in the way that neighbors are assembled for simultaneous application to a lookup table.

CAM-6 was designed primarily for running CA which employ traditional neighborhood formats, such as the "Moore" and "von Neumann" neighborhoods, in which one cell is updated as a function of more than one cell. Since this machine has many more cells than processors, cells within each module are processed sequentially. Thus new cell values cannot simply replace old values if the updating is to result in the same state that a simultaneous updating would produce—the old values must be retained as long as they may be needed in computing the new state of some cell. Because of this CAM-6 requires some buffering of cell values—neighborhood values sent to the lookup table are taken from this buffer (see Figure 7.1). For a 3×3 neighborhood, CAM-6 requires a 515-bit long buffer (2 lines plus 3 bits).

CAM-7 takes as its primary neighborhood format *partitioning cellular automata* (see Section 2.3). In this format, space is subdivided into disjoint subsets of cell bits. Lattice gas models are naturally described using this format: each site is updated independently of all the others, and then data is transferred between sites. Since each bit appears as part of only one site, the new values can immediately replace the old ones—no buffering such as was done in CAM-6 is needed. This format has a simpler hardware realization than traditional formats, and allows an enormous range of neighbor choices (as will be explained below).

Thus a CAM-7 step actually consists of two parts: an updating of all elements of the current partition, and a regrouping of data bits to form a new partition. The elements of the partition are just the 16-bit cells, each of which is updated by applying its value to a lookup table and storing the 16-bit result back into the cell. The partition is changed by shuffling bits between cells—how this is done is at the heart of CAM-7's design.

We take advantage of the fact that the plane-module—the elementary "chunk" of CAM-7's space—is much larger than a single cell. The data within one module can be shifted relative to the data in a second module by simply changing the place where we start scanning the data in the first module. Bits are shuffled between cells by shifting entire bit-planes, and this is accomplished by writing to registers that control where the next scan should begin within each plane-module. Since no time is stolen from the updating to accomplish these shifts, we refer to them as "instant shifts." In CAM-7, neighbors are gathered together by instant shifts.

To avoid complications associated with inter-module communication, consider first how these instant shifts work in a space of size 512×512×16. Each of the 16 modules consists of one 64K×4 DRAM chip plus a semi-custom controller chip. Given a horizontal and a vertical offset, the controller chip will take care of all of the details: it just has to read the nybbles of the memory chip in an order corresponding to a version of the plane that is shifted (with wraparound) by the given horizontal and vertical offsets. A four-bit pipeline inside the controller chip permits horizontal shifts that aren't a multiple of four. Thus the 16 bit-planes can be arbitrarily shifted relative to each other between one scan of the space and the next. As each cell is scanned, the 16 bits that come out at a given instant are applied as inputs to a lookup table, and the result is written back to the planes.[7]

The only point remaining to be explained is how the instant-shift process works when the machine is configured so that each bit-plane consists of many plane-modules "glued" together edge-to-edge. What

[7] To save address setup time on the DRAM chips, the controller reads a 4-bit nybble from memory and then immediately writes a new value (computed from cells accessed slightly earlier) to that same location. This results in a shift in the physical location of the cells in memory, which is also compensated for by a scan-origin shift within the controller chips.

happens is that each module separately performs a shift as described above. The wraparound occurs within each module: cells that should have shifted out the side of one module and into the opposite side of the adjacent module have instead been reinjected into the opposite side of the *same* module. The positions of these cells relative to the edges of a module are exactly as they should be for a true shift: they are just in the wrong module. However, since all modules output corresponding cells at the same moment, each module can produce a truly shifted output by simply replacing its own output with that of a neighboring module when appropriate.

For example, consider CAM-7 running in its 16384×8192×16 configuration. Each of the 16 bit-planes in this configuration consists of 512 plane modules, each of which scans an area 512×512. Now suppose we want to shift one of the bit planes 50 positions to the left. Each of the rows within each of the plane modules is rotated (circularly shifted) 50 positions to the left by appropriately changing the order of accessing the cell memory. Each module's controller chip will produce as an overall output a 512×512 window onto its portion of the complete shifted plane in the following way: the first 462 cell values of each row will come from the plane module's own rotated data, while the last 50 values will be "borrowed" from the rotated data of the module to its right.

Vertical gluing of bit-planes is achieved in a similar fashion. That is, the controller chip first glues plane-modules together horizontally; the output of this gluing process is further multiplexed across vertically adjacent modules, yielding the final output. In this way, each module only needs to be connected (by a single bidirectional line) to each of its four nearest-neighbor modules, and any shift of up to 512 positions horizontally, 512 vertically, or any combination of these can be accomodated. Thus any 16 bits (one from each plane) in a 1025×1025 region can be brought together and used as the neighbors to be jointly sent to the lookup tables.[8]

Of course, if we construct rules where the same table-output value is sent to, say, all 16 planes, then by shifting the planes as described above we can implement not only the traditional neighborhoods but also any other neighborhood entailing up to 16 bits chosen in a 1025×1025 region around each cell. Thus, conventional (i.e., nonpartitioning) cellular automata with very wide neighborhoods can also be simulated on CAM-7, albeit at the cost of using planes and tables rather redundantly.

7.6.3 Input and output

The basic bus on CAM-7 is the *flywheel bus*, consisting of the final glued outputs of the plane-modules together with inputs to these same modules. The input and output buses are each 8192 bits wide on a full CAM-7 machine: when the machine is operating at its maximum clock rate, a new 8192-bit output word is produced every 40 nanoseconds, and new input words can be accepted at the same rate. Every bit of

[8] Such large neighborhoods are, for example, particularly useful in image processing.

cell memory in the machine is available to be examined and modified once during every step. External logic (even, if desired, floating-point processors) can be attached here. Depending on how CAM-7 is configured, input bits can be ignored (in favor of internally-generated new cell values), routed as inputs to the lookup tables, or sent directly to the planes.

Besides the two data lines (one for input and one for output) that it contributes to the flywheel bus, each plane-module also has a small number of control lines. Some of these control lines are bussed in bulk to all the modules; the others are merged together into a *control bus* of moderate width. Areas that can be accessed via the control bus include

- the lookup table (with auto increment after each read or write)
- the bit-plane (with auto increment after each read or write)
- various registers (located within the controller chip)

  the horizontal-offset register

  the vertical-offset register

  the horizontal-size register

  the vertical-size register

  the table-address source select register

  the plane-data source select register
- various counters (located within the controller chip)

  the address counter

the table-correlation counter

the table-output counter

Each plane-module is connected both to one data line and to one address line of a lookup table. During normal updating, the address line is fed sequentially with the glued output of the bit-plane, and the values appearing on the data line are written sequentially into the bit-plane as its new contents. This is, however, just one possible combination of table-address and plane-data sources—by writing to a module's "source select" registers, any of the following may be sent either as an address bit to the lookup table, or as a data bit to be written directly into the plane:

- the glued output for this plane
- the output for the plane lying 8 positions above or below this one
- the output from corresponding plane in the other half of the machine
- the flywheel-bus input for this plane-module
- one bit from the address counter
- a constant of zero
- the complement of any of the above

Notice that the table output doesn't appear in this list—it can only be sent to the plane.

By appropriately controlling the sources both for table addresses and for plane data we can, for example, run a step in which a constant value of 0 or 1 is sent to the table address while the plane data is not affected—the plane can even be shifted during this step, since the table is not needed for this. Thus one can run steps during which one or more address bits of the table are host-selected constants, analogous to the "phase" bits[80] used by CAM-6. This allows one to split a lookup table into several subtables, to be used during consecutive steps without having to download new tables. Of course downloading new tables isn't a great problem as long as all the tables are identical (or there are only a few different kinds), since all tables that are the same can be written simultaneously.[9]

Data is read from or written to either planes or tables by the host in a similar manner: a stream of bits is sent to or from the module associated with the data. For planes, the horizontal- and vertical-offset registers are used not only during steps, but also to control where the data-bits sent by the host to the plane should go. For tables, each plane-module controls one bit of the address of a table, and is told by the host which bit of its internal address counter should be shown to its table, to control where data-bits go.

Note that these internal address counters are not provided solely

[9] If more flexibility in rewriting tables is needed, a number of microprocessors (say one for every 64 plane modules) could be added to the design. They could each store a selection of tables, and download them under the command of the host. They could also be useful in generating initial values for the cell states.

for loading tables; they can also be used during cell-updating, clocked by the 40ns system clock. By addressing a table with some counter bits one can, for instance, provide spatial parameters to CA rules (cf. the rotation algorithm in Section 7.6.8) or perform on-the-fly testing of tables.

7.6.4 Data analysis

Each plane module contains a number of counters that are used for real-time data analysis, error detection/correction, or both.

Table outputs are always counted (number of ones in each output). An analysis step can be performed by having some planes remain unchanged (or just shift) while the corresponding table outputs are being counted. For example, if a plane is being used to store a spatial parameter (such as an obstacle in a fluid-flow experiment) the associated table output is not needed for updating, and may be programmed for data analysis and counted. If there aren't enough such "free" tables, or if the analysis requires a different neighborhood than the updating, separate analysis steps may be interleaved between updating steps by rewriting tables.

CAM-7 can be operated as two half-machines—table outputs are continuously compared between corresponding parts of the two halves and the number of differences is counted by the *table correlation counters*. Space and time autocorrelations can be accumulated by running two versions of the same system simultaneously, with a constant space or time shift between them. Since both the number of differences between two corresponding table outputs and the number of ones output by each table separately are counted, the number of occurences of each of the four possible pairs of binary outputs can be computed. The fact that the sum of the two separate counts plus the correlated count should be even acts as a consistency check for detecting counter errors. If exactly the same system is run in both halves of the machine, the correlation counters detect updating errors.

Note that all counters are double-buffered, and can be read at any time by the host without affecting a step that is in progress.

7.6.5 Error handling

Like CAM-6, each CAM-7 machine will constitute a "building block" from which one can build much larger machines. For example, eight such blocks used together will have two giga-bytes of cell-state memory and will perform one-and-a-half trillion rather powerful cell-bit updates every second. While there are no inherent architectural limits on how many CAM-7's can be hooked together, there is a practical problem which grows as more and more CAM "blocks" are added, namely, error handling. Because of the built-in analysis capabilities described in the previous section, and additional hardware consistency checks, it will be possible to discover and recover from hardware errors.

Since tables are not supposed to evolve in time, it is relatively straightforward to test whether or not a table contains an error. We can usually detect table errors by performing an analysis step during which all tables are addressed by counter bits—we simply count the number of ones in all table outputs. As long as correlated pairs of tables contain the same rule we can simultaneously perform a more detailed check by comparing table outputs. Alternatively, we can have the host perform a verify-write of all tables, in which the old contents is read and compared with what the host is writing.

Cell-memory is tested by each plane-module during every step. About 22 checksum bits, reflecting the number of ones last written and their positions, are compared to corresponding checksums performed on the data subsequently read. Changing any bit of the configuration will, on the average, change about half of the checksum bits. By dividing all possible 512×512 configurations evenly into more than 10^6 different classes, these checksums make the chance of an undetected plane-memory error very small.

Hard errors, caused by bad components, can be tested for whenever any error is detected. If we run an occasional analysis step during which we test tables, bad chips should always be noticed quickly.

Soft errors, in which memory bits are typically changed, are principally caused by alpha particles. Modern commercial memory chips, which constitute most of CAM-7, are inherently quite reliable: even with absolutely no provision for error correction it should be possible to run this machine with 16384 memory chips for several days at a time without any errors. Thus for a single CAM-7 it may be perfectly practical in most cases to simply detect errors, and rerun an experiment if any occur. In fact for many statistical mechanical experiments, such as fluid flow past obstacles, a rare error in which a bit is dropped doesn't matter at all, and so we only need to rewrite incorrect tables and obstacles, and watch out for hard errors.

For longer runs, or for large machines built out of many CAM-7's, if we want to guarantee exactly correct operation it is probably most practical to use each machine as two correlated half-machines, both running the same experiment. Since the chance of two different plane-modules both experiencing a soft error during the same step is extraordinarily small (expected perhaps once in 10^{16} steps for a single CAM-7 machine) we can assume that one out of every correlated pair of plane-modules will always be correct. Planes that were updated incorrectly are fixed by using data from the correct twin, and incorrect tables are simply rewritten. Notice that to correct a plane-module, data doesn't even have to be physically moved from one module to its twin—we can simply run the next step with the correct module providing the input for the tables in both halves of the machine.

Given an error, there remains the problem of deciding which of the pair of correlated plane-modules is incorrect. For plane errors, we rely on the internal checksums maintained by the plane-modules to tell us which module to fix. Otherwise we make use of one further facility provided by the hardware in order to quickly and reliably find the error—even if it's a transient one that didn't change the contents of a table. Whenever table comparisons disagree, both the original contents of the cell where the error occured and the updated value are latched by the controller chips. By examining this information, the host can tell which of the planes was updated incorrectly.

7.6.6 Three-dimensional operation

When a single CAM-7 is operating in its 512×512×512×16 configuration, it is of course the fact that all plane-modules are updating the same position at the same time that allows information from one layer to be directly available for use by adjacent layers. In terms of plane-modules, one can think of this configuration as being 512×512×8192, i.e., 8192 deep in the third dimension. We prefer, however, to think of 512 "layers" each consisting 16 consecutive planes, since the outputs from each stack of 16 planes go to common lookup tables.[10]

Each plane-module in this 8K stack is connected to the module 8 positions above it and to the one 8 positions below it—a total of four wires (input and output above and below) time-shared between all 256K of the cell-bits on each module. Each module has several choices for what it sends as an address to its associated lookup table. It can of course send its own glued output. It can also send the glued output of

[10] External logic connected to the flywheel bus inputs and outputs can of course group these planes arbitrarily. For example, floating point processors might use them as multi-hundred-bit cells, each containing several floating point numbers that can be separately shifted to change the neighborhood. Since CAM processes each plane-module serially, these floating-point calculations could be pipelined—the delay between starting and finishing processing a cell could be lengthy, as long as a new cell value is completed every 40ns.

the plane 8 positions above or below itself. These three choices make three-dimensional operation straightforward.

For example, each 16-bit cell could be thought of as encoding the contents of a 2×2×2 cube having 2 bits at each site. The top eight bits in the cell (i.e., those belonging to the top eight planes in this 16-plane layer) would correspond to the top of the cube, the other eight to the bottom of the cube. After updating the cube according to some rule, let's say that we want to switch to a partition in which the corners of four adjacent cubes become the new cubes. To accomplish this, we will select for each lookup table input the output of the plane module 8 positions above—this is equivalent to shifting all of the plane da'a 8 positions down. Data from the bottoms of one layer of cubes now appear as inputs to the same tables as the tops of the next layer. We must now shift the planes corresponding to the various corners of the old cubes so that the data from four adjacent corners are shifted together. If we've been careful about what order within the cell the results of the first step were placed, we can even use the same rule on these new blocks. If we want different rules on the two partitions, we can of course rewrite the tables before each step. As we alternate between these two partitions, we can avoid a net motion of the cubes by alternately shifting the plane data up and down while moving the blocking back and forth in the other two directions as well.

Just as we could simulate the Moore and von Neumann neighborhoods in two dimensions, we can simulate nearby-neighbor interactions in three dimensions. For example, let's consider a rule that calls for the center cell, its 6 nearest neighbors, and the center cell in the "past" (i.e., the value the center cell had one step before), with two bits of state for each neighbor. We simply get two bits from the layer above, two from below, and the rest from the current layer (for a total of 16 bits). Our tables should each produce seven 2-bit copies of the new value for the center-cell, plus one copy of the present value (which will be used as the past by the next step). Four of the copies of the center cell will be shifted one position (north, south, east, and west). One will be visible only to the layer above, one only to the layer below, and the last to the current layer. The lookup tables can now calculate the updated values, and the process can be repeated. Other neighborhoods (for instance, the twelve second-nearest neighbors, or the eight third-nearest neighbors) can all be similarly implemented.

Notice that bits coming from above and below mask the corresponding bits from the current layer—the bit from the current layer no longer appears as an input to this layer's lookup table. You might worry that some bits could become completely hidden and not available as part of the neighborhood of any table, but this is never the case. The masked bit can simply be made visible 8 positions down within the current layer, masking another bit which is already visible as part of the neighborhood for the next layer.

What about rules that need more than 16 bits of input? By using some of the bit planes to store intermediate values, rules that need more bits of input can be synthesized as a composition of completely arbitrary 16-input/16-output logical functions. Taking advantage of the strong coupling between the two halves of each CAM-7 machine[11] one can readily synthesize rather large neighborhoods (up to 32 bits or more) by rule-composition. Note, however, that such compositions can entail, in the worst case, an exponential slow-down as the number of neighbors increases.

7.6.7 Display

Being able to display the state of our system in real time provides important feedback as to whether or not everything is working as expected, and what parts of the system are doing something interesting that should be investigated more closely.

Two dimensional display is not much of a problem for CAM-7, since this machine can provide its data in the correct format for a color monitor. This machine can even, if desired, scan its data in the correct format for an interlaced display—since each cell is updated independently of all others the rows can be scanned in whatever order you choose.

For a complete 16384×8192 display, we could cover an enormous wall with 512 color monitors (more if several CAM's are connected), each of which would show a 512×512 patch using 64K different colors.

[11] Recall that any bits from the 16-bit cell in one half can be substituted as table address sources for the corresponding bits in the other half. Machines connected via inputs on the flywheel bus are similarly strongly coupled.

Of course it might be more practical to use only one monitor (or just a few), and shift the data to move the window around. Using interlaced displays and a one-line buffer, 1024×1024 or even 2048×2048 regions could be viewed on a single monitor.

Since all of the neighbors that would be used for a cell-update are available simultaneously, it is a simple matter to display a function of the neighborhood, rather than the neighborhood itself. For example, in a fluid-mechanics experiment you might want to show only the smoke particles that trace the flow. Going a step further, part of the machine's resources could be devoted specifically to constructing the image to be displayed. For example, one half of the machine could do the experiment while the other half could monitor the first half, accumulating time-average data for the display.

CAM-7 realizes a three-dimensional system as a stack of two-dimensional layers, each of which can be viewed exactly as discussed above. In its 512×512×512×16 configuration, it would take 512 color monitors to see all layers at once; on the other hand, a single monitor would be enough to see any part of the cube, by shifting the data appropriately (now in three dimensions). Outputs from groups of layers could be combined (e.g., summed, OR'ed, etc.) and shown in a similar manner (still without any external frame buffer). You could even display a sum down through the entire machine—a sort of X-ray.

Suppose we would like to see slices through the cube perpendicular to the plane of our two-dimensional slices. This, and any other 90 degree rotation of the cube about its x, y, or z axis is easily accomplished by CAM using a simple split-and-shift algorithm.[12] Because of the instant shifts available along the bit-planes, rotations about one of the axes can be accomplished in a fraction of a second; rotations about the other two axes would take several seconds.

Such rotations would be particularly useful in conjuction with a display that provides a more natural format for CAM's 3-D ouput.

7.6.8 A true three-dimensional display

A true three-dimensional display (imagine a translucent cube hanging in mid-air and observable from within a wide angle) is achievable in a relatively straightforward manner. To illustrate the considerations involved, we will describe one particular technique.

Let us first construct a one-bit output for each of CAM-7's 512 layers; in this way, we obtain the equivalent of 512 TV-signal sources, all broadcasting in parallel. We would like to make up a cube out of these 512 TV frames, by literally stacking them in a third dimension like a deck of cards; as it turns out, it will be expedient to view the resulting "deck" from the top edge rather than from the front side.

Now, construct an array of 512×512 light emitting diodes; each row

[12] To rotate a square image, you can first split it into quarters, then shift the four quarters horizontally or vertically until they have each been shifted to a position 90 degrees clockwise of where they started. Each quarter is similarly rotated, and then each eighth, etc., until you reach the level of a single cell. Cells don't look any different when rotated, so you're done.

of LED's is driven by the outputs of a 512-bit, serial-in, parallel-out shift register with latched outputs (the equivalent of 32 74F673 chips). In turn, the shift registers are fed with the above TV sources, and their outputs latched at the end of every scan line. Thus the collection of 512 lines produced in parallel by CAM-7's 512 layers will have been captured as an two-dimensional LED picture; this picture, which lies orthogonally to the "cards of the deck," will last about 30usec before being replaced by the next picture, corresponding to the next scan line.

Every time a new LED picture is ready we want to display it somewhat below the previous one, so that starting from the top edge of the deck for the first line of the TV frame we will end up at the bottom edge with the frame's last line. This sweeping movement of the LED array is easily achieved by optical means—in a way similar to that demonstrated with success at BBN[68]. That is, the array will be viewed reflected on a thin-membrane mirror stretched over a loudspeaker. The speaker itself will be driven with a 60-Hz sawtooth wave, in sync with CAM-7's internal scan; the resulting slight changes in curvature of the mirror will make the LED-array's image sweep through a sequence of focal planes.[13]

Finally, to avoid filling the three-dimensional display with too much

[13] Note that in the BBN setup the performance of the system is limited by the available data rate (since the images to be optically multiplexed are generated by drawing vectors on a CRT) rather than by the optical arrangement. CAM-7, on the other hand, has a real-time data rate of 120 gigabits per second, which is more than sufficient to take full advantage of this arrangement.

data, some selective staining techniques may be appropriate, much as in microscopy. For instance, surfaces can be made visible by simulating "light" within the system: this would consist of particles that travel invisibly in a given direction and light up when they cross a surface (defined by an appropriate local condition).

7.7 Applications

In addition to statistical mechanical applications that are becoming known (fluid dynamics, Ising spin systems, optics, seismic waves, etc.) CAM-7 should be valuable for a number of less obvious applications.

For example, the structure of CAM-7 seems ideal for certain types of image processing; in particular, for certain "retina-like" tasks where the information contained in detailed two-dimensional images arriving in rapid succession is analyzed and preprocessed in real-time by algorithms that are in the main local and uniform, in order to supply a more "brain-like" post-processor with a much smaller amount of pre-digested data.

Each layer could run a different rule, each involving—if desired—rather widely scattered neighbors. Using the 3-D connections, with camera input going to the first layer, we could do some consecutive steps of image-processing in a pipelined manner—the output of one layer supplying the input for the next.[14] By custom wire-wrapping the flywheel-bus outputs and inputs, a much more complicated pipeline

[14] Since each layer can have a different rule stored in its look-up table, CAM-7 as a whole is a true *multiple-program, multiple-data* machine.

could be achieved. For example, the output of one layer could become the input to several other layers, which could then lead to other layers; there could be further splits and merges, data following a shorter path could be time-correlated with data following a longer path, etc.

CAM-7 could be used for digital logic simulations in two or three dimensions. Since bit-planes can be made to shift by large amounts between steps, signal speeds would not necessarily be limited to one cell per step. CAM-7 could also be used as a testbed for ideas about using cellular automata VLSI chips as "soft circuitry." For example, given a chip that runs a simple 2-D rule such as LOGIC (Section 4.4), one could download a pattern of wires and gates to a chip, and have it simulate the circuit fast enough to actually be used in placed of the target circuit itself.

In general, this machine should be useful in a range of simulation and modeling tasks involving systems which have an appropriate local structure.

Bibliography

[1] ALADYEV, Viktor, "Computability in Homogeneous Structures," *Izv. Akad. Nauk. Estonian SSR, Fiz.-Mat.* **21** (1972), 80–83.

[2] AMOROSO, Serafino, and Y. N. PATT, "Decision Procedures for Surjectivity and Injectivity of Parallel Maps for Tessellation Structures," *J. Comp. Syst. Sci.* **10** (1975), 77–82.

[3] BANKS, Edwin, "Information Processing and Transmission in Cellular Automata," *Tech. Rep. MAC TR-81*, MIT Project MAC (1971)

[4] BENIOFF, Paul, *J. Stat. Phys.* **22** (1980) 563; **29** (1982) 515.

[5] BENIOFF, Paul, "Quantum Mechanical Hamiltonian Models of Discrete Processes That Erase Their Own Histories: Application to Turing Machines," *Int. J. Theor. Phys.* **21** (1982) 177–201.

[6] BENIOFF, Paul, "Quantum Mechanical Hamiltonian Models of Computers," *New Techniques and Ideas in Quantum Measurement Theory* (Daniel GREENBERGER ed.), New York Academy of Sciences (1986), 475–486.

[7] BENNETT, Charles, "Logical Reversibility of Computation," *IBM J. Res. Develop.* 6 (1973), 525–532.

[8] BENNETT, Charles, "Thermodynamics of Computation," *Int. J. Theor. Phys.* **21** (1982), 905–940.

[9] BENNETT, Charles, and Geoff GRINSTEIN, "Role of Irreversibility in Stabilizing Complex and Nonenergodic Behavior in Locally Interacting Discrete Systems," *Phys. Rev. Lett.* **55** (1985), 657–660.

[10] BENNETT, Charles, Tommaso TOFFOLI and Stephen WOLFRAM, "Cellular Automata 86," *Tech. Memo LCS-TM-???*, MIT Lab. for Comp. Sci. (1987).

[11] BENNETT, Charles, Norman MARGOLUS and Tommaso TOFFOLI, "Bond Energy Variables For Spin Glass Dynamics," submitted for publication.

[12] BERLEKAMP, Elwyn, John CONWAY, and Richard GUY, *Winning Ways For Your Mathematical Plays*, vol. 2, Academic Press (1982).

[13] BURKS, Arthur (ed.), *Essays on Cellular Automata*, Univ. Ill. Press (1970).

[14] CALIFANO, Andrea, Norman MARGOLUS and Tommaso TOFFOLI, *CAM-6 User's Guide*; *and* Kenneth PORTER, *CAM-6 Hardware Manual*, Systems Concepts, 55 Francisco St., San Francisco 94133 (1987).

[15] CODD, E. F., *Cellular Automata*, Academic Press (1968).

[16] COX, J. Theodore, David GRIFFEATH, "Recent Results For the Stepping Stone Model," *University of Wisconsin Math Department preprint.*

[17] CREUTZ, Michael, "Deterministic Ising Dynamics," *Annals of Physics* **167** (1986), 62–76.

[18] DEUTSCH, David, "Quantum Theory, the Church-Turing Hypothesis, and Universal Quantum Computers," *Proc. Roy. Soc.* (1985).

[19] D'HUMIÈRES, Dominique, Pierre LALLEMAND, and T. SHIMOMURA, "Lattice Gas Cellular Automata, a New Experimental Tool for Hydrodynamics," Preprint LA-UR-85-4051, Los Alamos National Laboratory (1985).

[20] FARMER, Doyne, Tommaso TOFFOLI, and Stephen WOLFRAM (eds.), *Cellular Automata*, North-Holland (1984).

[21] FELLER, William, *An Introduction to Probability Theory and Its Applications*, vol. I, 3rd ed., Wiley (1968).

[22] FEYNMAN, Richard, "Simulating Physics with Computers," *Int. J. Theor. Phys.* **21** (1982), 467-488.

[23] FEYNMAN, Richard P., "Quantum Mechanical Computers," *Opt. News* **11** (1985).

[24] FREDKIN, Edward, private communication.

[25] FREDKIN, Edward, and Tommaso TOFFOLI, "Conservative Logic," *Int. J. Theor. Phys.* **21** (1982), 219–253.

[26] FRISCH, Uriel, Brosl HASSLACHER, and Yves POMEAU, "Lattice-Gas Automata for the Navier-Stokes Equation," *Phys. Rev. Lett.* **56** (1986), 1505–1508.

[27] GACS, Peter, and John REIF, *Proc. 17-th ACM Symp. Theory of Computing* (1985), 388-395.

[28] GARDNER, Martin, "The Fantastic Combinations of John Conway's New Solitaire Game 'Life'," *Sc. Am.* **223**:4 (April 1970), 120–123.

[29] GREENBERG, J., and S. HASTINGS, "Spatial Patterns for Discrete Models of Diffusion in Excitable Media," *SIAM J. Appl. Math.* **34** (1978), 515.

[30] HAYES, Brian, "The Cellular Automaton Offers a Mcdel of the World and a World Unto Itself," *Scientific American* **250**:3 (1984), 12–21.

[31] HARDY, J., O. DE PAZZIS, and Yves POMEAU, "Molecular Dynamics of a Classical Lattice Gas: Transport Properties and Time Correlation Functions," *Phys. Rev.* **A13** (1976), 1949–1960.

[32] HEDLUND, G. A., K. I. APPEL, and L. R. WELCH, "All Onto Functions of Span Less Than or Equal To Five," Communications Research Division, working paper (July 1963).

[33] HEDLUND, G. A., "Endomorphism and Automorphism of the Shift Dynamical System," *Math. Syst. Theory* **3** (1969), 51–59.

[34] HERRMANN, Hans, "Fast Algorithm for the Simulation of Ising Models," Saclay preprint no. 86-060 (1986).

[35] HOLLAND, John, "Universal Spaces: A Basis for Studies in Adaptation," *Automata Theory*, Academic Press (1966), 218–230.

[36] KADANOFF, Leo, "On two levels," *Physics Today* **39**:9 (September 1986), 7–9.

[37] KIMURA, M., G. WEISS, "The Stepping Stone Model of Population Structure and the Decrease of Genetic Correlation With Distance," *Genetics* **49** (1964), 561–576.

[38] KIRKPATRICK, Scott, C.D. GELATT Jr., M.P. VECCHI, "Optimization by Simulated Annealing," *Science* **220** (1983), 671–680.

[39] KNUTH, Donald, *The Art of Computer Programming*, vol. 2, Seminumerical Algorithms, 2nd ed., Addison-Wesley (1981).

[40] LANDAUER, Rolf, "Irreversibility and Heat Generation in the Computing Process," *IBM J. Res. Devel.* **5** (1961), 183–191.

[41] LANDAUER, Rolf, "Computation and Physics," to appear in Foundations of Physics (1986).

[42] LANDAU, L., E. LIFSHITZ, *Mechanics*, Pergamon Press (1960).

[43] LIKHAREV, K. K., "Classical and Quantum Limitations on Energy Consumption in Computation," *Int. J. Theor. Phys.* **21** (1982), 311–326.

[44] MANDELBROT, Benoit, *The Fractal Geometry of Nature*, W. H. Freeman (1982).

[45] MARGOLUS, Norman "Physics-like Models of Computation," *Physica* **10D** (1984), 81–95.

[46] MARGOLUS, Norman, Tommaso TOFFOLI, and Gérard VICHNIAC, "Cellular-Automata Supercomputers for Fluid Dynamics Modeling," *Phys. Rev. Lett.* **56** (1986), 1694–1696.

[47] MARGOLUS, Norman, "Quantum Computation," *New Techniques and Ideas in Quantum Measurement Theory* (Daniel GREENBERGER ed.), New York Academy of Sciences (1986), 487–497.

[48] MARGOLUS, Norman, "Partitioning Cellular Automata," in preparation.

[49] MARUOKA, Akira, and Masayuki KIMURA, "Conditions for Injectivity of Global Maps for Tessellation Automata," *Info. Control* **32** (1976), 158–162.

[50] MARUOKA, Akira, and Masayuki KIMURA, "Injectivity and Surjectivity of Parallel Maps for Cellular Automata," *J. Comp. Syst. Sci.* **18** (1979), 47–64.

[51] MÉZARD, M., "On the Statistical Physics of Spin Glasses," *Disordered Systems and Biological Organization* (E. BIENENSTOCK et al., ed.), Springer-Verlag (1986), 119–132.

[52] MINSKY, Marvin, *Computation: Finite and Infinite Machines*, Prentice-Hall (1967).

[53] ORSZAG, Steven, and Victor YAKHOT, "Reynolds Numbers Scaling of Cellular-Automaton Hydrodynamics," *Phys. Rev. Lett.* **56** (1986), 1691–1693.

[54] PACKARD, Norman, and Stephen WOLFRAM, "Two-dimensional Cellular Automata," *J. Stat. Phys.* **38** (1985), 901–946.

[55] PEARSON, Robert, "An Algorithm for Pseudo Random Number Generation Suitable for Large Scale Integration," *J. Computat. Phys.* **3** (1983), 478–489.

[56] PERES, Asher, "Reversible Logic and Quantum Computers," *Phys. Rev. A* (Dec. 1985).

[57] PERES, Asher, "Measurement of Time by Quantum Clocks," *Am. J. Phys.* **48** (1980), 552.

[58] POMEAU, Yves, and P. RÉSIBOIS, *Phys. Rep.* **19** (1975), 63.

[59] POMEAU, Yves, "Invariant in Cellular Automata," *J. Phys.* **A17** (1984), L415–L418.

[60] POROD, W., R. GRONDIN, D. FERRY, and G. POROD, "Dissipation in Computation," *Phys. Rev. Lett.* **52** (1984), 232; comments by Ç. H. Bennett, P. Benioff, T. Toffoli, and R. Landauer *Phys. Rev. Lett.* **53** 1202.

[61] REITER, Carla, "Life and Death on a Computer Screen," *Discover* (August 1984), 81–83.

[62] RESSLER, Andrew L., *The Design of a Conservative Logic Computer and a Graphical Editor Simulator*, (Master of Science Thesis), Massachusetts Institute of Technology (1981).

[63] RICHARDSON, D., "Tessellation with Local Transformations," *J. Comp. Syst. Sci.* **6** (1972), 373-388.

[64] ROSENBERG, I., "Spin Glass and Pseudo-Boolean Optimization," *Disordered Systems and Biological Organization* (E. BIENENSTOCK et al., ed.), Springer-Verlag (1986), 327–331.

[65] SALEM, James, and Stephen WOLFRAM, "Thermodynamics and Hydrodynamics of Cellular Automata," *Theory and Applications of Cellular Automata* (Stephen WOLFRAM ed.), World Scientific (1986), 362–366.

[66] SANDER, Leonard, "Fractal growth processes," *Nature* **322** (1986) 789–793.

[67] SHANNON, Claude and W. WEAVER, *The Mathematical Theory of Communication*, Univ. of Illinois Press (1949).

[68] SHER, Lawrence and C. D. BARRY, "The Use of an Oscillating Mirror for 3-Dimensional Display," *New Methodologies in the Study of Protein Configuration*, (edited by T. T. WU), Van Nostrand (1985), Chapter 6.

[69] SMITH, Alvy, "Cellular Automata Theory," *Tech. Rep. 2*, Stanford Electronic Lab., Stanford Univ. (1969).

[70] STANLEY, H. Eugene, and Nicole OSTROWSKY, *On Growth and Form*, Martinus Nijhoff (1986).

[71] TOFFOLI, Tommaso, "Cellular Automata Mechanics," *Tech. Rep. 208*, Comp. Comm. Sci. Dept., The Univ. of Michigan (1977).

[72] TOFFOLI, Tommaso, "Computation and Construction Universality of Reversible Cellular Automata," *Journal of Computer Systems Science* **15** (1977), 213–231.

[73] TOFFOLI, Tommaso, "Integration of the Phase-Difference Relations in Asynchronous Sequential Networks," *Automata, Languages, and Programming* (Giorgio AUSIELLO and Corrado BÖHM ed.), Springer-Verlag (1978), 457–463.

[74] TOFFOLI, Tommaso, "Bicontinuous Extension of Reversible Combinatorial Functions," *Maths. Syst. Theory* **14** (1981), 13–23.

[75] TOFFOLI, Tommaso, "Reversible Computing," *Automata, Languages and Programming* (DE BAKKER and VAN LEEUWEN eds.), Springer-Verlag (1980), 632–644.

[76] TOFFOLI, Tommaso, "CAM: A High-Performance Cellular-Automaton Machine," *Physica* **10D** (1984), 195–204.

[77] TOFFOLI, Tommaso, and Norman MARGOLUS, "The CAM-7 Multiprocessor: A Cellular Automata Machine," *Tech. Memo LCS-TM-289*, MIT Lab. for Comp. Sci. (1985).

[78] TOFFOLI, Tommaso, "Cellular Automata as an Alternative to (Rather Than an Approximation of) Differential Equations in Modeling Physics," *Physica* **10D** (1984), 117–127.

[79] TOFFOLI, Tommaso, and Norman MARGOLUS, *Invertible Cellular Automata*, in preparation.

[80] TOFFOLI, Tommaso, and Norman MARGOLUS, *Cellular Automata Machines: A New Environment for Modeling*, MIT Press (1987).

[81] TUCKER, Jonathan, "Cellular Automata Machine: The Ultimate Parallel Computer," *High Technology* **4**:6 (1984), 85–87.

[82] TURING, Alan, "On Computable Numbers, With an Application to the Entscheidungsproblem," *Proc. London Math. Soc., ser. 2,* **43** (1936), 544-546.

[83] ULAM, Stanislaw, "Random Processes and Transformations," *Proc. Int. Congr. Mathem.* (held in 1950) **2** (1952), 264-275.

[84] VAN DYKE, Milton, *An Album of Fluid Motion*, Parabolic Press (1982).

[85] VICHNIAC, Gérard, "Simulating Physics With Cellular Automata," *Physica* **10D** (1984), 96–115.

[86] VICHNIAC, Gérard, "Cellular Automata Models of Disorder and Organization," *Disordered Systems and Biological Organization* (BIENENSTOCK et al. eds.), Springer-Verlag (1986), 1–20.

[87] VON NEUMANN, John, *Theory of Self-Reproducing Automata* (edited and completed by Arthur BURKS), Univ. of Illinois Press (1966).

[88] WITTEN, Thomas, and Leonard SANDER, *Phys. Rev. Lett.* **47** (1981), 1400.

[89] WOLFRAM, Stephen, "Statistical Mechanics of Cellular Automata," *Rev. Mod. Phys.* **55** (1983), 601–644.

[90] WOLFRAM, Stephen, "Universality and Complexity in Cellular Automata," *Physica* **10D** (1984), 1–35.

[91] WOLFRAM, Stephen, "Computation Theory of Cellular Automata," *Commun. Math. Phys.* **96** (1984), 15-57.

[92] WOLFRAM, Stephen, "Random-Sequence Generation by Cellular Automata," *Adv. Applied Math.* **7** (1986), 123–169.

[93] WOLFRAM, Stephen (ed.), *Theory and Applications of Cellular Automata*, World Scientific (1986).

[94] ZAIKIN, A., and A. ZHABOTINSKY, *Nature* **225** (1970), 535.

[95] ZUREK, W. H., "Reversibility and Stability of Information Processing Systems," *Phys. Rev. Lett.* **53** (1984) 391.

[96] ZUSE, Konrad, *Rechnender Raum*, Vieweg, Braunschweig (1969); translated as "Calculating Space," *Tech. Transl. AZT-70-164-GEMIT*, MIT Project MAC (1970).