Nanotechnology **14** (2003) 469–485 PII: S0957-4484(03)39229-3

Laying out circuits on asynchronous cellular arrays: a step towards feasible nanocomputers?

Ferdinand Peper, Jia Lee, Susumu Adachi and Shinro Mashiko

Communications Research Laboratory, Nanotechnology Group, 588-2 Iwaoka, Iwaoka-cho, Nishi-ku, Kobe, 651-2492, Japan

Received 8 July 2002, in final form 1 January 2003
Published 20 March 2003
Online at stacks.iop.org/Nano/14/469

Abstract
Opinions differ widely as to the type of architecture most suitable for achieving the tremendous performance gains expected with computers built by nanotechnology. In this context little research effort has gone into asynchronous cellular arrays, an architecture that is promising for nanocomputers due to (1) its regular structure of locally interconnected cells, and (2) its asynchronous mode of timing. The first facilitates bottom-up manufacturing techniques like directed self-assembly. The second allows the cells' operations to be timed randomly and independently of each other, mitigating the problems accompanying a central clock, like high power consumption and heat dissipation. The advantages of asynchronous timing notwithstanding, it makes computation less straightforward. Attempts to compute on asynchronous cellular arrays have therefore focused on simulating synchronous operation on them, at the price of more complex cells. Here we advance a more effective approach based on the configuration on an asynchronous cellular array of delay-insensitive circuits, a type of asynchronous circuit that is robust to arbitrary delays in signals. Our results may be a step towards future nanocomputers with a huge number of autonomously operating cells organized in homogeneous arrays that can be programmed by configuring them as delay-insensitive circuits.

M This article features online multimedia enhancements

1. Introduction

Though the recent progress in the development of logic gates and simple circuits on molecular scales [1–13] (for a short review see [14]) brings nanocomputers a step closer to reality, it is still an open question as to what computer architectures are most suitable for them [15–24]. The prevailing von Neumann architecture, characterized by the separation of memory and processing resources, poses challenges in particular when superimposed onto nanotechnology [15, 19, 22, 23]. Apart from its irregular structure, which complicates cost-effective manufacturing on molecular scales, it also faces problems with its global connectivity: at higher integration densities, computers based on the von Neumann architecture are not gate limited but rather interconnection limited [22].

Suffering less from such problems are architectures for nanocomputers based on *cellular arrays*. Locally connected in homogeneous patterns, such arrays contain vast numbers of identical simple cells. The regular structure of an array of cells not only reduces design efforts to a few parts that can be reused many times in a design, it also opens up the possibility of bottom-up molecular manufacturing techniques. Though it is far from trivial to bulk manufacture cellular arrays on nanometre scales, the task is considerably easier than manufacturing the irregular structures found in von Neumann computers. By employing chemical techniques to produce components in large quantities and piece them together in arrays by directed self-assembly, it may be possible to construct computers with up to the order of Avogadro's number of simple identical cells. Cellular arrays range from coarse-grained systems, like the propagated instruction processor (PIP) [25] and the reconfigurable architecture workstation (RAW) architecture [26], to fine-grained systems, like systolic arrays [27], field programmable gate array (FPGA) based systems [22] and cellular automata, the last model of this list being focused upon in this paper.

A *cellular automaton* [28–30] is an array of cells, each of which is a finite automaton [31]. The cells interact locally in accordance with a set of rules that are designed such as to produce certain global behaviour, like general-purpose computation [28]. We will denote this model by the more general term 'cellular arrays', as the term emphasizes the model's regular structure and it is often used in an implementation-oriented context. A cellular array designed with the specific aim of nanometre-scale computation in mind is Biafore's cellular array [17]. It implements Fredkin's billiard ball model [32], which conducts reversible general-purpose computation by the ballistic interactions of idealized billiard balls with each other. The cells consist of quantum-dot devices, organized such that electrical charge is transferred under the control of optical clock signals of different wavelengths supplied in a specific order. Another cellular array architecture aiming for efficient implementations by nanotechnology is the *cell matrix* [19]. In this model each cell, containing a memory of less than 100 bytes, can be programmed to calculate a specific function, like a NAND, an XOR, a one-bit full adder, a wire etc.

The above cellular arrays are timed synchronously, requiring the delivery of a central clock signal to each cell. A synchronous mode of timing gives rise to a wide array of problems, like high power consumption and, associated with it, heat dissipation (more details in section 3). These problems tend to get worse at increased integration densities.

It thus makes sense to consider asynchronous cellular arrays [33], a computation model that pairs a homogeneous structure with a mode of operation in which all cells conduct simple operations timed randomly and independently of each other. Though attractive for attaining efficient physical realizations, this asynchronous mode of operation brings up the question of how to actually compute on such cellular arrays. The few attempts to do so [33–37] have focused on simulating a timing mechanism on an asynchronous cellular array to force the cells into synchronicity, and then utilizing well-established methods to compute synchronously. This is inefficient, not only because the simulation of the timing mechanism requires increased complexities of the cells, but also because the computing methods used are sequential to begin with, and thus fail to exploit the massive parallelism of cellular arrays (more on the disadvantages of this method in section 7).

Here we show a more elegant approach based on an asynchronous cellular array that conducts asynchronous computations directly without resorting to synchronous methods. Requiring just four bits of memory to store its state, each cell in the cellular array needs access to only its own four bits and one bit of each of its four neighbouring cells. To enable the cellular array to conduct the same class of computations as that possible on conventional computers, it requires as few as nine rules describing interactions between its cells.

Our approach exploits the properties of so-called *delay-insensitive circuits*, a type of circuit that allows arbitrary delays in signals without this being an obstacle to its correct operation. Delay-insensitive circuits not only offer many advantages in themselves over synchronous circuits [38–42], they also combine well with asynchronous cellular arrays. Their robustness to signal delays creates substantial freedom in laying them out on asynchronous cellular arrays, since the requirement no longer holds that signals must arrive at certain places at times dictated by a central clock, as in synchronous circuits. This takes away concerns about variations in the operational speed of cells and considerably simplifies the design of configurations representing circuit elements laid out on asynchronous cellular arrays.

Computers based on asynchronous cellular arrays may lead to tremendously increased performance, a computational potential especially useful in applications requiring massive parallelism, like simulations of particle systems in physics, simulations of networks of neural cells, genetic algorithms and artificial intelligence.

Section 2 of this paper defines the (two-dimensional) asynchronous cellular array model used. We also show how signals are transmitted over the cells. Section 3 starts with an overview of asynchronous circuits, and in particular delay-insensitive circuits. We then describe how they operate and show an implementation on the asynchronous cellular array of a set of circuit primitives from which arbitrary delay-insensitive circuits can be constructed. Section 4 deals with the interactions between the cells. Interaction rules are defined, and it is shown how they can be used to drive the primitives in section 3 when implemented on the cellular array. We then construct a configuration of cells by which signals can be made to cross on the cellular array, this guaranteeing that arbitrary circuits can be laid out on the cellular array. This section finishes with the construction of a one-bit memory on the cellular array. Section 5 aims to close the gap between cells and circuits on one hand and higher-order program structures on the other hand. To this end, a for-loop is constructed using the one-bit memory. Section 6 discusses implementation details and section 7 finishes this paper with conclusions and a discussion.

2. Asynchronous cellular arrays

A cellular array is a d-dimensional array of identical cells (this paper assumes $d = 2$), each of which can be in one of a finite number of states, denoted as the integers $0, 1, \ldots, n-1$, where n is the number of states. Each cell can read the states of itself and of its neighbouring cells, and change its own state in accordance with these states. Such a state change is called a *transition*, and an expression describing an allowable transition is called a *transition rule*. The set of cells whose state can be accessed by a cell is called the *neighbourhood* of the cell. A frequently used neighbourhood is the *von Neumann* neighbourhood: it consists of a cell's nearest orthogonal neighbours. Another type of neighbourhood sometimes encountered is the *Moore* neighbourhood, which consists of all nearest cells adjacent to a cell including the diagonal cells. A Moore neighbourhood is usually employed on *totalistic* cellular arrays, i.e., models in which the *number* of neighbouring cells in certain states, rather than the states themselves, form the basis of transitions. Certain desired behaviour is imposed on a cellular array by setting its cells in proper states and defining transition rules that lead to state changes corresponding to this behaviour.

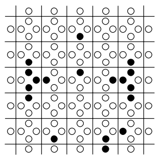


Figure 1. Cellular array consisting of cells, each with 16 states that are encoded by four bits. A filled (black) circle indicates a bit with the value 1 and an open (white) circle indicates a 0-bit.

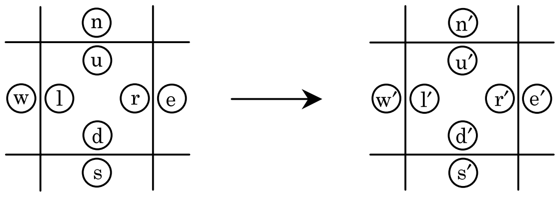


Figure 2. Transition rule. The variables in the circles denote states of the corresponding bits, which may be 0 or 1. The left-hand side of the rule (the part left of the arrow) gives the states of a cell's bits before a transition, and the right-hand side gives the states after the transition.

For example, one may design a cellular array that can conduct the same class of computations as those possible on a conventional computer [28], or conduct operations by which configurations of cells in certain states are copied to other parts of the cellular space (self-reproduction) [43–45], or both [28, 29, 46, 47]. Cellular arrays in which all cells undergo transitions at the same time are called *synchronous*. They are the most widely studied type of cellular array.

When transitions of the cells occur at random times, independent from each other, we obtain *asynchronous cellular arrays*. This paper employs a recently proposed asynchronous cellular array called a *self-timed cellular automaton (STCA)* [35]. An STCA is a two-dimensional asynchronous cellular array of identical cells, each of which can be in one of 16 states. The state of a cell is encoded by four bits that are positioned along the cell's sides, where a filled (black) circle indicates a bit valued 1 and an open (white) circle a bit valued 0. Figure 1 depicts part of an array of such cells.

Each cell can undergo transitions, which take place in accordance with transition rules that operate on the four bits of the cell and the nearest bit of each of its four neighbours. Assuming that the bit states are denoted by u, u', r, r', d, d', l, l', n, n', e, e', s, s', w, $w' \in \{0, 1\}$, a transition rule $f(u, r, d, l, n, e, s, w) = (u', r', d', l', n', e', s', w')$ can be depicted as in figure 2, where the condition $(u, r, d, l, n, e, s, w) \neq (u', r', d', l', n', e', s', w')$ holds.

When the bit pattern of a cell matches the left-hand side of a transition rule (the part left of the arrow in figure 2), the cell will undergo a transition, but the timing of the transition is random. If there is no transition rule whose left-hand side matches the cell's bit pattern, the cell remains inactive. Though transitions of cells are timed randomly and independently of each other, they are subject to the condition that two neighbouring cells never undergo transitions simultaneously. This ensures that two neighbouring cells will not attempt to set the two bits common to them to different values at the same time. Figure 3(a) gives an example of a transition rule that, operating on a pattern of one 1-bit and seven 0-bits, moves

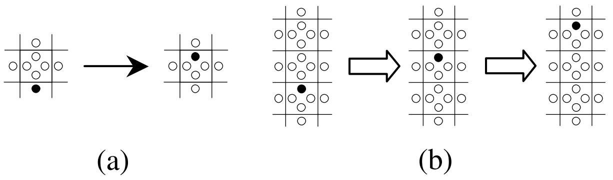


Figure 3. (a) Transition rule, and (b) its application to a configuration of cells, giving rise to a signal successively moving two cells towards the north. A filled (black) circle indicates a 1-value and an open (white) circle a 0-value. The transition rule operates on eight bits: on the four bits of a cell itself, as well as on the nearest bit of each of its four neighbours.

the 1-bit one cell to the north. The rule is only applied to cells of which the four bits and the four neighbouring bits exactly match the pattern of eight bits on the left-hand side of the rule in figure 3(a). Applied twice, the transition rule in figure 3(a) gives rise to the sequence of configurations in figure 3(b) in which a 1-bit moves into the northern direction along an uninterrupted area of cells. The rotation-symmetric and reflection-symmetric equivalents of transition rules may also serve as transition rules. This allows the above transition rule to be used for transmitting signals in directions towards the south, east, or west as well. Section 6 gives a more detailed description of the algorithms in accordance with which transitions take place on an STCA.

Having defined the transmission of signals, we describe in the next sections how they can be operated upon by delay-insensitive circuits laid out on an STCA. This way of computing may be characterized as *collision based computing* [48], a paradigm in which compact patterns, our signals, wander around in a structureless space and smash into other compact patterns, which in our case are the elements of delay-insensitive circuits.

3. Delay-insensitive circuits

A delay-insensitive circuit is a circuit whose correctness of operation is unaffected by arbitrary delays of its signals in the circuit elements and the interconnection wires. Operations in delay-insensitive circuits are driven by signals: each circuit element is inactive unless it receives an appropriate set of input signals, after which it processes the signals, outputs appropriate signals and becomes inactive again. Since delay-insensitive circuits do not require a central clock signal, they belong to the larger class of *asynchronous circuits*, which have several advantages over synchronous circuits [38, 39, 41, 42] (see also [49] for an informal discussion):

(1) *Circuitry for distributing the clock signal not required.* The area required for circuitry to distribute the clock signal increases with the number of components in the circuit, and this area may become very high for molecular-scale integration densities. Apart from significantly saving on circuit area, the absence of circuitry to distribute the clock signal also improves the homogeneity of the system—a great advantage for nanometre-scale fabrication.

(2) *Less energy consumption and heat dissipation.* Only those parts of an asynchronous circuit that are active draw power, whereas in synchronous circuits all elements

have to switch with the clock. Coming for free in asynchronous circuits, this automatic power saving feature would require special circuitry to be implemented in synchronous circuits, circuitry that in its turn consumes the power and chip area.

(3) *Problems with the timing of signals disappear. Clock-skew*, the time differences at which different parts of a circuit receive the clock signal, and *race conditions*, the failure of signals to reach their destinations within a clock cycle, are especially serious at higher integration densities and high clock frequencies; there are techniques to deal with them, but these tend to consume much circuit area and power. As wire delays have improved by only 20% in each new integrated circuit process generation [50]—much less than the 150% improvement in gate delays each generation—it was only a matter of time before it became impossible to reach an entire die within a single clock cycle. At this rate, less than 10% of die area will be reachable once feature sizes reach 60 nm [50].

(4) *Less noise and electromagnetic interference.* Unlike in synchronous systems, there are no distinct peaks in the RF spectrum, resulting in less interference and noise.

(5) *Insensitivity to physical implementations and conditions.* The proper operation of an asynchronous circuit is guaranteed under a wide variety of physical implementations and physical conditions. Variations in the timing of signals related to these factors do not affect the correctness of a circuit's operation, especially if the circuit is delay insensitive. This also implies more freedom in laying out circuits.

(6) *Average rather than worst-case performance.* An asynchronous circuit operates as fast as switching times of devices allow, rather than being limited by the slowest parts of the circuit, which in synchronous systems limit the global clock rate.

(7) *Modularity.* Asynchronous circuits can be subdivided into modules that can be designed independently and combined without considerations of timing restrictions. The operational correctness of a module does not depend on other modules, as long as basic design constraints are obeyed. Modularity guarantees that circuit elements can be rearranged easily into various circuits.

Since the above advantages are felt even more strongly at higher integration densities, there is increasing interest in asynchronous circuits, even though they have disadvantages too, like the much less available design, testing and manufacturing infrastructure and expertise as compared to synchronous systems. Moreover, asynchronous circuits often require additional hardware to avoid *hazards*, non-monotonic changes in the value of signals, which may cause unexpected circuit behaviour. As synchronous circuits allow signals to arrive out of tone as long as they settle down within a clock signal, they have been considered easier to design than asynchronous circuits, whose signals must be correct the first time they arrive at their destination. Hazards may be less of an issue, however, for implementations of asynchronous circuits in some technologies different from solid-state electronics, for example, technology based on molecular mechanisms, such as the molecular cascades [51] mentioned in section 7.

Another drawback of asynchronous circuits is the overhead caused by their signalling protocol. The usual way to initiate an action in an asynchronous circuit is to send a *request* signal. The circuit then signals the completion of the action by an *acknowledge* signal. This way of signalling, called *handshaking*, requires many connections and feedback connections to make circuit modules work together seamlessly. As a result, extra time for operations and extra circuitry is required, which diminishes the benefits of asynchronous circuits in practice. The delay-insensitive circuits used in this paper, however, require less feedback connections than usual, since they allow multiple signals at a time in each of their interconnection wires [52], implying that it is not necessary for a circuit element to receive an acknowledge signal before outputting signals to its output wires. Though such a signalling method has never been considered before, as it is impractical for solid state electronics-based systems, it is easily—and most easily—implemented in cellular array-based systems.

Asynchronous circuits come in many flavours (see [38] for an overview and [39] for an overview with a history of asynchronous circuits). The most general class is formed by asynchronous circuits that make use of timing assumptions both within the circuit and in the interaction between circuit and the environment, for example assumptions on the boundedness of delays [53]. This class of signals is the least robust to unexpected behaviours. The second class constitutes *self-timed* circuits [54]. The elements of self-timed circuits make no assumptions on the timing of communications *between* them. This delay-insensitive mode of communication contrasts with the mode of communication *within* the elements, which are strictly regulated: for example, wires inside elements may be required to have bounded or negligible delays. In the third class, *speed-independent* circuits [55], operations of circuit elements may be subject to unbounded delays, but wires between elements have zero or negligible delays. These circuits are in practice very similar [38] to circuits in the fourth class, *quasi-delay-insensitive* circuits [56], which are delay insensitive except that they require a so-called *isochronic fork* [57, 58], a fan-out element of which all output branches have the same delay. The fifth class constitutes the circuits with which we started this section, *delay-insensitive* circuits. The most robust of all asynchronous circuits due to their tolerance to delays of both circuit elements and wires, delay-insensitive circuits are very suitable for implementation on asynchronous cellular arrays. This type of circuit has its roots in the work by Clark and Molnar in the 1960s on *macromodules* [59, 60], developed at Washington University in St Louis, a project resulting in a set of easily interconnected hardware modules from which computer systems can be readily assembled. Those not initiated in the subtleties of electronics are able to construct working computers from these modules—a powerful demonstration of the composition benefits of delay-insensitive circuits. Follow-up of this work is in [40, 41, 61–64] among others. Also contained in this class are *micropipelines* [65], which are designed to be an asynchronous alternative to synchronous elastic pipelines—pipelines in which the number of data can vary—but they also serve as a powerful method for implementing general computations. As they use a *bundled data* protocol—a bundle of bounded-delay data wires, each transmitting 1 bit, combined with two delay-insensitive control

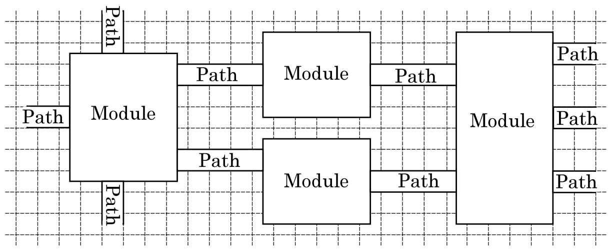


Figure 4. Modules connected by paths on a cellular array.

wires, one for requests and one for acknowledgements—they are not delay insensitive in the strict sense, so they are sometimes considered a class on their own [38].

Unlike in synchronous circuits that use binary valued signals, information in delay-insensitive circuits is often represented by the presence of signals on particular wires. To transmit a binary value, for example, two wires are used: a signal on one of the wires denotes the value 0, and a signal on the other wire denotes the value 1. Called *dual-rail encoding* [66], this way of encoding allows a signal on only one of the two wires at a time. The absence of signals on both wires indicates that no information is being transmitted. Though dual-rail encoding requires more wires than bundled data encoding, we use dual-rail encoding in this paper, because a bundled data scheme demands that data signals arrive earlier than control signals, which is hard to guarantee in implementations on asynchronous cellular arrays.

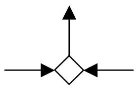
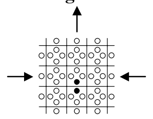
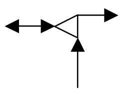
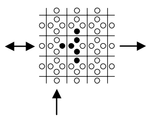
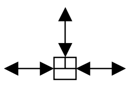
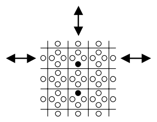
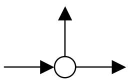
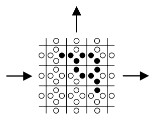
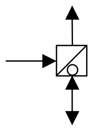
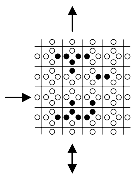
Delay-insensitive circuits consist of *modules*, which are elements with a finite number of input and output wires and a finite number of states. When receiving certain signals from the input wires, a module conducts an operation, as a result of which it may change its state and output certain signals on its output wires. Modules can be organized in a hierarchy, ranging from a module constructed from networks of modules, which may be as complex as a delay-insensitive computer, to modules so simple that they cannot be subdivided any further, so-called *primitive* modules. Realized in a cellular array by configurations of cells in certain states, modules connect to each other by *paths* (see figure 4), uninterrupted areas of cells over which signals are transmitted from a source module to a destination module.

A path of cells plays the same role in asynchronous cellular arrays as a wire in solid-state circuits. We then define a *signal* as the change of state or value of part of a path, directed from one of its ends (the *source*) to the other end (the *destination*). There can be more than one signal on each path, but signals will never interfere with each other on a path, i.e., different signals are not fused into a single signal and every single signal does not spontaneously split into more than one signal. Once a signal is transmitted from a source, it cannot be withdrawn, and it will head for its destination. The presence of other signals will never prevent it from reaching its destination, though they may delay it by a finite amount of time.

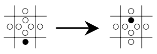
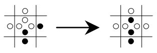
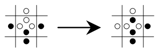
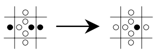
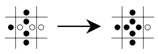
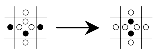
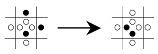
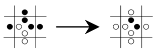
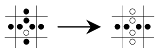
Once an input signal reaches its destination module, it is assimilated and the module conducts an operation, which usually results in the output of signals on one or more of its paths. The module may have to wait, though, for other input signals to arrive from different paths if its operation requires it. In this case, the input signal is called *pending*. The transmission of signals over paths and their processing by modules may undergo any finite time delay, without this having any consequences for the correct operation of the circuit. In other words, there are no time constraints on signals. There are a few rules governing the assimilation of signals by modules. Two successive input signals on one path to a module are always interspersed in time by an output signal produced by the module as a response to the first signal. This effectively prevents the module from using successive input signals from the same path for one operation. A different situation occurs when a module is presented with two input signals from different paths, and it can only process one at a time. In this case the module may arbitrarily choose which signal to process first. It is then said to *arbitrate* between its input signals. Systematic overviews of the conditions under which delay-insensitive circuits operate can be found in [40] and [52].

As in synchronous systems, in which any arbitrary Boolean circuit can be constructed from a primitive like the NAND-gate, for delay-insensitive circuits too there is a set of primitives from which any arbitrary circuit can be constructed. Such a set is called *universal*. For practical purposes, universality can be considered as the constructability from the set of primitives, of circuits that can conduct the same types of computation as possible on conventional computers (see also discussion in [52]). Boolean gates are not universal primitives for delay-insensitive circuits, because they lack the functionality to deal with every possible temporal ordering of events on their input wires [57, 67]. A set of primitives for delay-insensitive circuits must provide both universal logic functionality as well as timing functionality, the latter being necessary to make up for the loss of the clock. Keller proposed a set of delay-insensitive primitives and showed that, given an arbitrary module's specification in terms of its states' changes and its output signals for every allowable combination of input signals, the module can be constructed from the set of primitives, which implies that the set of primitives is universal [40]. Ebergen proposed an alternative set of delay-insensitive primitives [63], and proved by using a formal system called a *trace language* that the set is universal [68]. Patra [41] used Keller's work as the base for yet another universal set of delay-insensitive primitives. An extensive listing of primitives available in delay-insensitive systems is given in [69]. Most relevant to the current paper is an overview of the primitives of Keller [40] and Patra [41], and the proposal of a novel universal set of primitives based on them, given by Lee *et al* [52]. We use the set of primitives of Lee *et al*, because it goes particularly well with asynchronous cellular arrays. This set is listed in box 1 (left-hand side). Unlike delay-insensitive primitives proposed in the past, these primitives may have bi-directional interconnections, i.e., paths that can be used for both input as well as for output, albeit not at the same time. Moreover, a path may contain multiple signals at the same time, provided they all move in the same direction. Under these conditions at most three interconnections are required for input and output to each of the primitives, as shown in [52], which substantially simplifies the simulation of the primitives on asynchronous cellular arrays.

The *merge* primitive (see box 1), proposed in [52], merges two input streams into a single output stream. It differs from

Symbol		**Configuration**
	**Merge:** An operator with two input paths and one output path that redirects signals arriving from the input paths to the output path. The simultaneous arrival of one signal at each of the two input paths gives rise to two successive output signals.	
	**Input/Output Multiplexer:** An operator with one input path, one output path, and one bi-directional path. An input signal on the input path is output to the bi-directional path and an input signal on the bi-directional path is output to the output path.	
	**Symmetric Join:** An operator with three bi-directional paths, of which two act as input paths and one as output path. An input signal at one of the paths keeps pending until there is an input signal from one more path, after which both signals are joined, resulting in a signal output to the remaining path.	
	**Fork:** An operator with one input path and two output paths. Upon assimilating a signal from its input path, it produces one signal on each of its two output paths.	
	**Arbitrating Test and Set:** An operator with a bi-directional path, an input path, and an output path. It has two states, 1, the initial state, and 0, the state reached when it assimilates an input signal from its bi-directional path. Upon receiving a signal from its input path, the operator outputs a signal to its output path if in state 1, and it outputs a signal to its bi-directional path if in state 0, immediately after which it sets its state back to 1. If there are input signals at both the input and the bi-directional path at the same time, the operator processes one signal, then the other in an arbitrary order.	

Box 1. Primitive operators of delay insensitive circuits. The filled circles in the configurations indicate 1-values, and the open circles indicate 0-values.

1.		Rule for Signal Transmission (1), copied from figure 3(a)		
2.		Rules for Merge (2, 3)	3.	
4.		Rules for Input / Output Multiplexer (4, 5)	5.	
6.		Rules for Symmetric Join (6, 7)	7.	
8.		Rules for Fork (8) and Arbitrating Test and Set (9)	9.	

Box 2. Transition rules for simulating delay insensitive circuits.

the merge primitive in literature (see for example [40, 60]) at a crucial point: unlike the conventional merge, which demands mutually exclusive inputs, our merge allows input signals that arrive at its input paths simultaneously. It deals with such input by forwarding one of the input signals first to the output path, while keeping the input signal at the other input path pending until it is ready to handle the signal. Since our model allows multiple signals to be on a path at a time, it is not necessary to check whether the output path already contains a signal before a new signal can be output to it. Our merge has the advantage over the conventional merge in that it can be used [52] to construct a *sequencer*—a module for arbitrating the passage

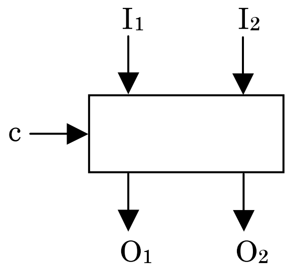


Figure 5. A sequencer module. An input signal on wire I_1 (respectively I_2) together with an input signal on wire c but without an input signal on wire I_2 (respectively I_1) are assimilated, resulting in an output signal on wire O_1 (respectively O_2). If there are input signals on both I_1 and I_2 at the same time as well as an input signal on *c*, then only one of the signals on I_1 and I_2 (possibly chosen arbitrarily) is assimilated together with the signal on *c*, resulting in an output signal on the corresponding O_1 or O_2 wire. The remaining input signal will be processed at a later time, when a new signal is available on wire *c*.

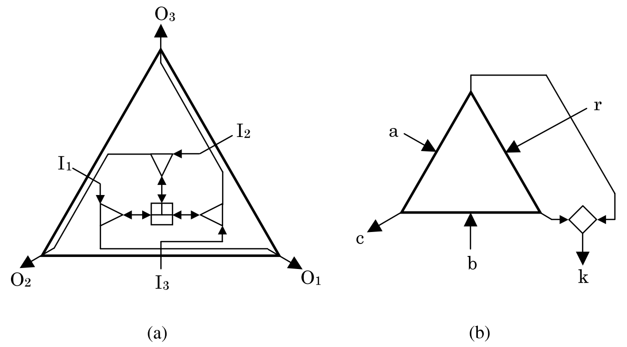


Figure 6. (a) A TRIA module constructed from one symmetric join and three input/output multiplexers (adjusted from [52]). Upon receiving an input signal from each of the two wires I_i ($i \in \{1, 2, 3\}$) and I_j ($j \in \{1, 2, 3\} \setminus \{i\}$), it outputs a signal to the wire O_{6-i-j}. Though the symbol for the TRIA is similar to that of the input/output multiplexer, they can be easily distinguished between by the positions, numbers and types of the wires. (b) A resettable join module constructed from a TRIA and a merge. When receiving one input signal each from wires a and b, the module outputs a signal to wire c. An input signal pending on either a or b is redirected to output wire k when an input signal is assimilated from reset wire r.

of signals on two paths through it (see figure 5)—that runs in *without-busy-waiting* mode—a mode that does not require a signal to continuously idle around in the intra-modular circuit checking for input. As sequencers in which only conventional merge primitives are used cannot run in this mode [41], they require the underlying hardware to continuously switch, a potential source of heat dissipation.

The *input/output multiplexer*, proposed in [52], is used to transform one input path and one output path into one bi-directional path. It is a prerequisite for any scheme that employs bi-directional paths.

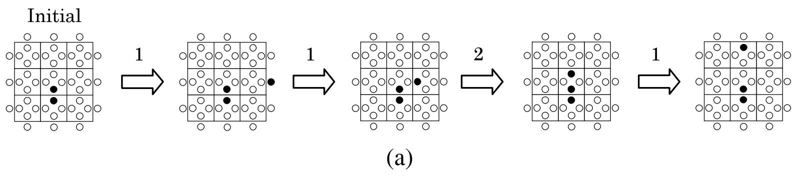
The *symmetric join* primitive, proposed in [52], produces an output signal upon receiving one input signal from each of two different paths, and can be thought of as synchronizing its two input signals. It differs from the join primitive in the literature (see for example [40]) in that it has three bi-directional paths, rather than the two input paths and one output path of the conventional join. This equips it with a functionality much richer than that of a conventional join: when an input signal is pending on one of the paths of a symmetric join, it is still undetermined at which of the remaining paths the output will emerge since this depends on which of the remaining paths the second input signal is received. The symmetric join can be used to construct modules like a *TRIA* [41] and a *resettable join* [69] (see figure 6), whereas a conventional join is insufficient for this.

The *fork* primitive is a fan-out element commonly used in delay-insensitive circuits [40, 60]. It produces one signal at each of its output paths for every signal it receives from its input path.

The *arbitrating test and set* is a primitive to test the presence of an input signal on its bi-directional path. Triggered by a signal to the input path, the module returns an output signal to the bi-directional path if there was a signal input to the bi-directional path, and it returns an output signal to the output path otherwise. An important condition to be fulfilled by the outside circuitry for the proper use of the arbitrating test and set is that no two subsequent signals may be input to the bi-directional path: they should always be interspersed by one output signal from this path. Every signal input to the bi-directional path thus gives rise to exactly one signal being output to this path, triggered by an input signal to the input path. Necessary for constructing a sequencer module (see figure 5), this primitive is used for arbitration between processes that want access to a shared resource, which is an important functionality in a massively parallel model like cellular arrays. Though the functionality of any arbitrary delay-insensitive circuit can be realized without the use of an arbitrating test and set primitive [40] or a sequencer module, only limited efficiency can be achieved when access of shared resources by parallel processes is required, since this can only be done by alternately activating and deactivating the processes. For the implementation of delay-insensitive circuits on our STCA, arbitration is necessary to ensure that signals on crossing paths can pass the crossing—a shared resource—without conflicts (see also [70] and section 4). The absence of arbitration would severely degrade the performance of circuits, up to the point that in the worst case only a single signal at a time would be able to run around in the whole circuit. The arbitrating test and set primitive is basically the same as the arbitrating test and set of Keller [40], except that it has one path less: one input path and one output path of Keller's arbitrating test and set is combined into one bi-directional path, to keep the number of paths of the primitive limited to three. Variations on this theme are also possible, for example the corresponding primitive in [52], called *reflexive arbitrating test and set*, combines the other input path and output path too, giving rise to only two bi-directional paths for the primitive. For the implementation on the STCA in this paper, however, it is most convenient to use three paths for the primitive.

4. Laying out delay-insensitive circuits on asynchronous cellular arrays

Any arbitrary delay-insensitive circuit can be realized in an STCA by using the cell configurations on the right of box 1 to represent the corresponding primitives on the left, and connecting them to each other by paths of cells. To make



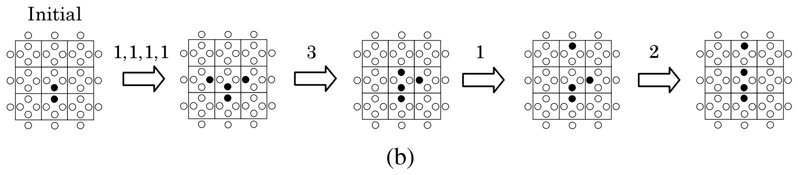
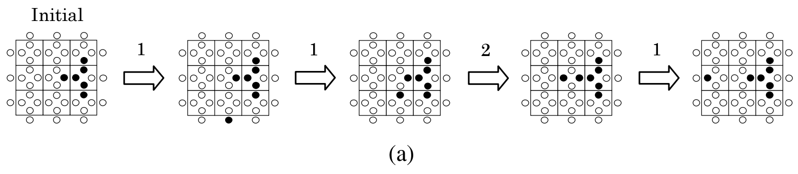


Figure 7. Sequence of configurations in which a merge receives (a) one signal and (b) two simultaneous signals. All signals are redirected towards the output path. The initial configuration without input signals is labelled as such. Each time a transition rule is used, its label in box 2 appears above the corresponding arrow.



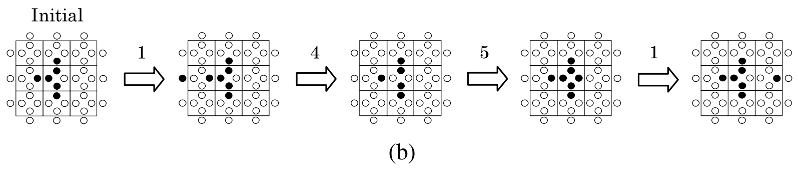
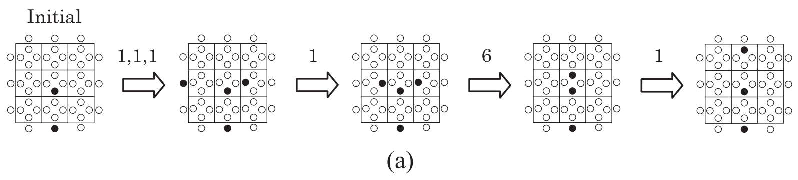


Figure 8. Sequence of configurations in which an input output/multiplexer operates (a) on a signal on its input path resulting in an output signal on its bi-directional path, and (b) on a signal input to its bi-directional path resulting in a signal on its output path.



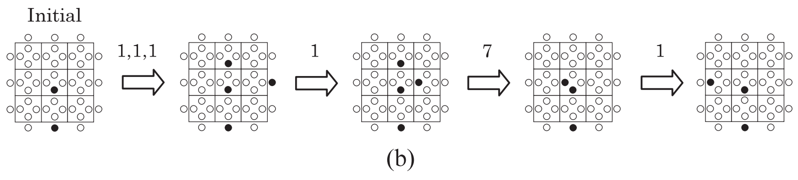


Figure 9. Sequence of configurations in which a symmetric join operates on two signals from (a) opposite paths, and (b) adjacent paths, in both cases producing one output signal.

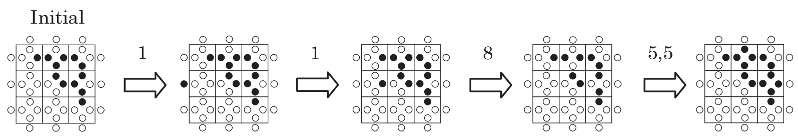
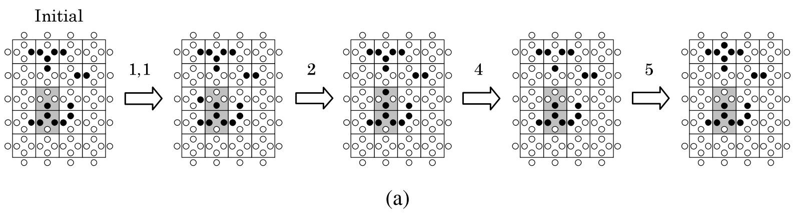


Figure 10. Sequence of configurations in which a fork receives one signal and outputs two signals.

these configurations behave like the primitives, we give a set of nine transition rules describing how cells interact with each other (see box 2).

Only when the configurations on the right of box 1 are presented with input signals do they undergo transitions, as the transition rules fail to match them otherwise. The configurations are thus stable in the absence of input signals.

The merge primitive, represented by a configuration of two adjacent 1-bits, processes a single signal in accordance with rule 2 in box 2, as in figure 7(a), and two simultaneous signals



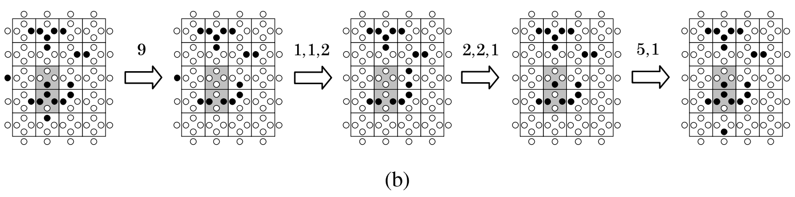
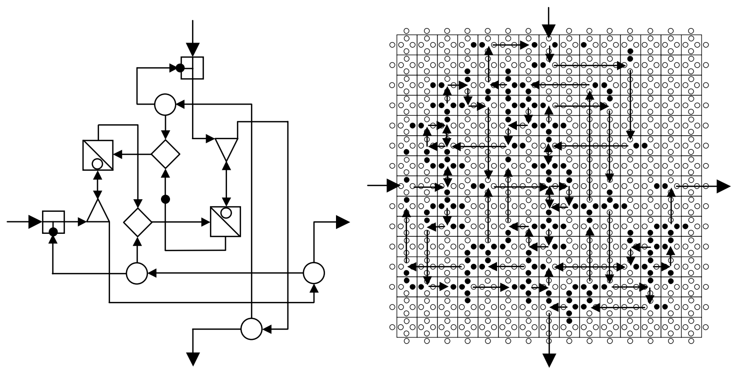


Figure 11. A sequence of configurations in which an arbitrating test and set operates on (a) a signal from its input path, resulting in a signal to the output path, and (b) an input signal from its bi-directional path (arriving first) followed by a signal from its input path, resulting in an output signal on the bi-directional path.



Delay-insensitive circuit scheme for crossing signals without collisions (left), and its implementation on the STCA (right). The small arrows in the cellular array denote paths via which signals can travel between primitives, whereas the big arrows denote input and output paths. Each of the two Symmetric Join primitives acts as a sluice to prevent a new input signal from entering the circuit as long as the circuit is still processing the previous input signal at the corresponding input path. A pending signal (denoted by a black blob at a Symmetric Join primitive) indicates that the circuit is ready to receive new input from the corresponding path. The circuit contains two Arbitrating Test and Set primitives that register the arrival of signals from the two input paths. A signal running around in a loop (denoted by a black blob in the circuit scheme) scans these primitives alternately and produces an output signal at the corresponding path if the primitive is in a state denoting the arrival of an input signal.

Box 3. Crossing signals on the cellular array. (For supporting online material (movie) see [71].)

in accordance with rule 3, as in figure 7(b). This primitive can also be used for right and left turns of signals.

The input/output multiplexer is represented by a configuration of three pairs of 1-bits. For a signal from its input path, as in figure 8(a), the configuration works as a merge primitive, outputting the signal on the bi-directional path towards the west, due to rule 2. If a signal is input to its bi-directional path at the west, as in figure 8(b), it passes right through the configuration, leaving it at its eastern side, in accordance with rules 4 and 5.

The symmetric join primitive is represented by a configuration of two 1-bits at the opposite exterior of a cell. If only one signal is input to this primitive, it is kept pending until a second signal enters, after which both signals are assimilated, producing one output signal. Rule 6 applies when the input signals originate from opposite paths (see figure 9(a)), whereas rule 7 applies when the input signals originate from adjacent paths (see figure 9(b)).

The fork configuration uses rule 8 in combination with rule 5 to process a signal from its input path. Passing through the sequence of successive configurations in figure 10, it produces one signal at the east and one at the north, though not necessarily simultaneously due to the asynchronous operation of the cellular array.

Finally, rule 9 is used for the arbitrating test and set primitive, which is represented as a configuration built up around a core—displayed as a shaded area in figure 11—that is basically an input/output multiplexer, but with additional functionality. In its normal state this core redirects signals arriving from its input path towards its output path using rules

2, 4 and 5. When a signal to the input path is preceded by a signal to the bi-directional path, however, as in the leftmost configuration in figure 11(b), the middle bit pair of the core is first set to 0 in accordance with rule 9, after which the signal from the input path follows a different route: due to rule 2 it is redirected into the core, coming from the north, after which it resets the middle bit pair of the core to 1 by rule 5 and leaves the primitive at its bi-directional path.

When realizing a delay-insensitive circuit by laying out primitives on the cellular array and connecting them to each other by paths, special care is required for paths that cross each other, since the cellular array lacks a third dimension via which crossings can be made. The cells at the intersection of a crossing are a shared resource, so arbitration is required to distribute this resource amongst the paths. To this end, we adapt the design of the busy-waiting sequencer in [41] to make it suitable for signal crossings. The resulting circuit scheme is given in box 3, together with its implementation on the cellular array. This configuration guarantees collision-free crossings, whatever the order in which signals arrive at the input paths. A more complicated and efficient version of this circuit, based on a without-busy-waiting sequencer [52], is given in [70]. Simulations conducted on a computer confirm the correct operation of the signal crossing (see online supporting material [71]). The use of the crossing configuration allows any particular delay-insensitive circuit to be implemented on the cellular array with the configurations on the right of box 1 as the base. This includes memory registers, circuits to do arithmetic and so on.

Finally, we construct a one-bit memory with the control circuitry required for read/write operations. Such a one-bit memory is proposed as a delay-insensitive primitive in [40] (where it is called an *S-module*), but we design its circuit here in terms of the lower-level primitives in box 1. The resulting implementation on the cellular array is given in box 4. All path crossings in this memory are collision free, and thus do not require the crossing configuration in box 3. The circuit design in box 4 can be carried over without problems to the framework of conventional delay-insensitive circuits, as each path in this circuit contains at most one signal at a time. A more complicated circuit design for the one-bit memory is in [41]. Simulations conducted on a computer confirm the correct operation of the one-bit memory (see online supporting material [72]).

5. Higher order program structures

An important issue for a nanocomputer will be—as for any computer—whether it lends itself to easy programming, and whether the huge amount of software available for current computers can be easily translated to run on it. Examples of software used to construct delay-insensitive hardware from program descriptions are Philip's Tangram compiler [73], Martin's Communicating Processes compiler [56] and Ebergen's system to translate specifications in trace language into circuits [63]. The Tangram compiler has been used to develop a pager with delay-insensitive logic, which not only resulted in only half the power consumption of comparable products, but also in substantially less RF interference. Martin used his compiler to design a computer with quasi-delay-insensitive logic. The above achievements indicate that delay-insensitive circuits can be generated from program descriptions, but this work tends to be directed towards circuit design rather than program design.

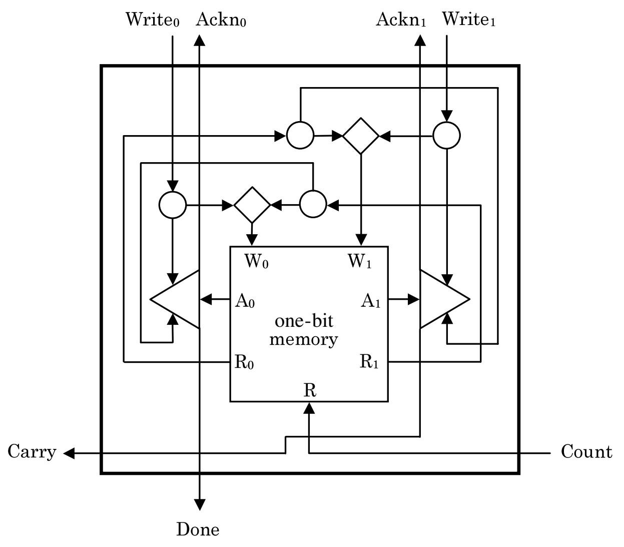
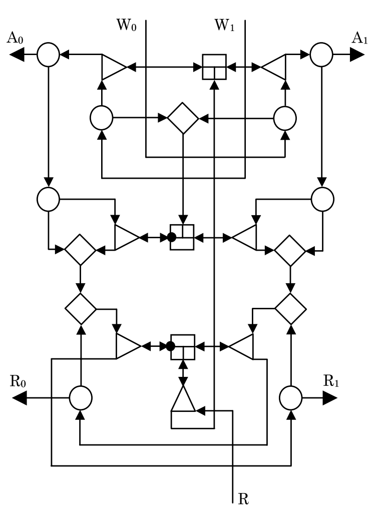
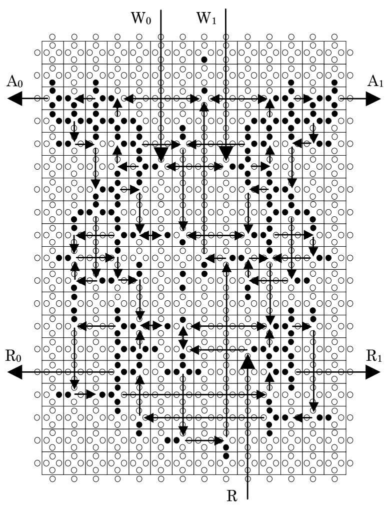


Figure 12. A one-bit counter constructed from a one-bit memory. To initialize the counter's memory, a signal is input to the Write$_0$ or Write$_1$ wire, giving rise to an acknowledgement on the Ackn$_0$ or Ackn$_1$ wire, respectively. A signal input via the Count wire flips the value of the counter's memory: if the value changes to 1, a signal is output to the Carry wire, otherwise it is output to the Done wire. Both initialization as well as updating of the one-bit memory takes place via the input wire W$_0$ or W$_1$ of the one-bit memory, so signals for initialization and updating are merged before being input to W$_0$ or W$_1$, respectively. To split the output signals emerging from the A$_0$ or A$_1$ wire, respectively, in accordance with whether they originate from initialization or from updating, two TRIA modules are used.

In this section we take a different approach and express higher order programming structures directly in terms of the delay-insensitive circuit primitives in box 1. Due to the modularity of delay-insensitive circuits, such programming structures can be easily combined in large programs. Our starting point is the one-bit memory in the previous section. Though Keller has used such a one-bit memory as a primitive [40] in combination with his other primitives to construct larger modules, his constructions are not intuitively straightforward, and hardly serve as a model for building delay-insensitive software. To stay a little closer to well known programming concepts, we outline how a *for-loop*, which is an important program structure, can be constructed from our primitives. We first construct a one-bit counter from a one-bit memory, then combine n one-bit counters into one n-bit counter, and finally construct a for-loop, using the counter for controlling the number of times the loop is run through. Though an implementation of the loop structure on the STCA is not given due to space considerations, the resulting cellular array configuration can be easily obtained by piecing together the configurations corresponding to the primitives and modules used in the loop structure.

A one-bit counter (figure 12) flips the contents of its bit each time it receives an input signal on the Count wire, producing an output signal on the Carry wire if the flipping operation changes the bit to 1, and on the Done wire otherwise. Also contained in the module is circuitry to initialize the bit to 0 or 1.





Delay-insensitive circuit scheme of a 1-bit memory (left), and its implementation on the STCA (right). The state of the memory is stored by two pending signals (indicated by black blobs in the circuit scheme) on the left or right input paths to two Symmetric Join primitives. A signal pending on the left input path denotes 0, the initial state, and a signal on the right denotes the state 1. Writing a 0- or 1-bit into the memory is done by sending a signal to one of the input paths W_0 or W_1 respectively. Due to this, the two pending signals storing the memory's state are removed, after which the actual write signal, which is temporarily kept in the remaining Symmetric Join primitive at the top of the circuit, is loaded in each of the two Symmetric Join primitives. This is accompanied by an output signal on either A_0 or A_1, depending on whether the write signal was W_0 or W_1, respectively, to acknowledge that the memory is ready to receive new input signals. Reading the memory is done by sending a signal to the input path denoted by R, which, after reading out the lower Symmetric Join primitive, results in a signal on either R_0 or R_1, depending on the state of the memory.

Box 4. A one-bit memory with its control circuitry. (For supporting online material (movies) see [72].)

The next step is to combine n one-bit counters into an n-bit counter (figure 13). The combined memories of the one-bit counters form an n-bit register into which a value can be written via the Write wires of the one-bit counters. Once the register is initialized, say to the value k, the counter will be decreased by 1 each time an input signal is received from the Count wire, giving rise to one signal output to the Done wire the first k times, and one signal output to the Carry wire the $k + 1$th time. The Dummy wire has no particular function here, but can be used to combine counters into bigger ones by connecting one counter's Dummy wire with another counter's Done wire.

The construction of the for-loop is given in figure 14. Writing k, the number of times the loop should be executed, into the n-bit counter gives rise to one acknowledging signal from each of the counter's memory bits. The resulting n acknowledging signals are joined into one signal that is used to activate the initialization of the loop. A signal emerging from the Done wire of the loop initialization signifies its end, and this signal is used to start the counter, which subsequently activates the body of the loop k times. Finally, a signal emerges from the End wire, which, apart from denoting the end of the loop's execution, can also be used to start up the next process.

Program structures are static, and as such they can be represented as delay-insensitive circuits. How about data? Though data are more dynamic [16], their configuration usually takes place in large blocks. Provided that the structure of the data is known in advance, such blocks can be represented as delay-insensitive circuits. An even more dynamic handling of data can be achieved in implementations on asynchronous cellular arrays, if delay-insensitive circuits can be configured dynamically on the cellular arrays, for example by using techniques resembling self-reproduction [28, 43–46] (see also the discussion in section 7). We will not pursue this issue in more detail here, as it is beyond the scope of this paper, but only note that in principle no insurmountable obstacles need to be expected.

6. Methods and implementation

Whereas there are well-established methods to design logic circuits in synchronous systems and to minimize the numbers of gates used, the design and optimization of delay-insensitive circuits is less straightforward. The methods to automatically generate delay-insensitive circuits from program descriptions [56, 63, 73], mentioned in section 5, are aimed at the use of primitives different from ours. We have no description language and translation software available yet for our set of primitives, so our design process is not guided by a systematic method, but by hands-on reasoning, which proved to be satisfactory for the design of the signal crossing circuit and the one-bit memory.

The implementations on the STCA of the primitives and sample circuits were verified to be correct by simulations on

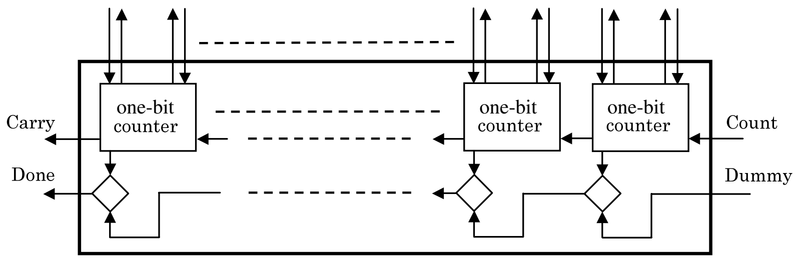


Figure 13. An n-bit counter constructed from n one-bit counters.

a PC with an Intel Pentium 4 processor running under Linux. The simulation program was written in the program language Java, which proved to be sufficiently fast for our purposes. The cellular array was updated in accordance with the following algorithm.

- Determine a cell c at random in accordance with a uniform probability distribution.
- Check whether the values of the eight bits associated with the cell (the cell's own four bits and one bit of each of its four neighbours) match the values of the bits on the left-hand side of a transition rule.
- If there is a match, update the bits associated with cell c in accordance with the matched transition rule; otherwise, do nothing.
- Repeat the above procedure.

This algorithm ensures that on average each cell is probed at the same rate. Simultaneous updating of two neighbouring cells, which is forbidden, is automatically excluded by this updating algorithm. Simultaneous updating of two non-neighbouring cells, though not done explicitly by the algorithm, is included in the set of behaviours it can simulate. For, as two non-neighbouring cells have no shared bits and thus can be updated independently, every scheme of updating non-neighbouring cells simultaneously can be expressed as a scheme of updating them sequentially. We conclude that the algorithm covers every allowed updating order of the cellular array, whether it be simultaneous updating of non-neighbouring cells or not.

When implementing the STCA cellular array physically, rather than simulating it as we did, a scheme will be required according to which simultaneous updating of two neighbouring cells is ruled out. To prove that such a scheme is possible in principle, we give an algorithm for this task. We extend each cell by a bit that can be written into by the cell itself and only read out by the neighbours of the cell. This bit, called the *a-bit*, indicates to neighbouring cells that the cell may become active and undergo a transition, thus preventing them from doing the same. The algorithm runs independently in each cell and is as follows:

(1) Set own a-bit to 0
(2) If the a-bit of any neighbour cell is 1, then go to 2
(3) Set own a-bit to 1
(4) Wait a random time
(5) If the a-bit of any neighbour cell is 1, then go to 1
(6) If the status bits associated with the cell match a transition rule's left-hand side, do the corresponding transition
(7) Go to 1.

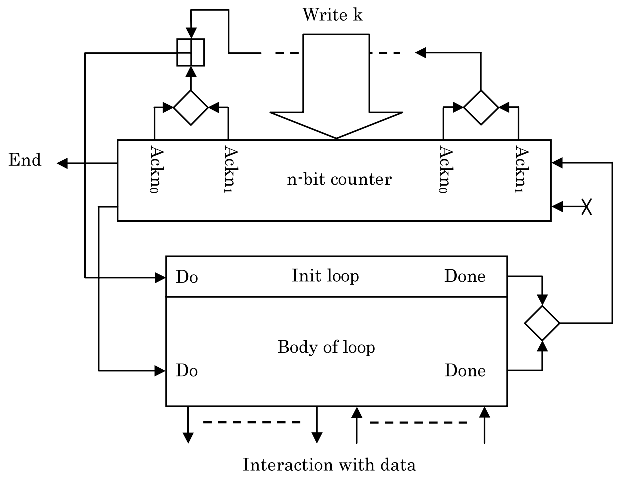


Figure 14. Construction of a for-loop. After writing the number of times (k) the loop should be executed in the n-bit counter, the loop is initialized by the acknowledge signals emerging from the writing operation. This is followed by the execution of the loop k times. A signal emerging from the End wire signifies the end of the loop's execution.

Though the a-bits of two neighbouring cells may be simultaneously 1 some of the time, at least one of the cells will reset its a-bit to 0 and forego the right to undergo a transition, due to line 5 of the algorithm. If both of two neighbouring cells reset their a-bits to 0, neither of them undergoes a transition and the whole procedure is repeated, until one of the cells passes line 5. Line 4 is used to prevent two neighbouring cells from getting phase locked to each other, i.e., prevent them from having to flip their a-bits repeatedly to 0 and 1 with neither of them getting beyond line 5. By using this algorithm, it is not necessary to update the eight state bits associated with a cell in synchrony during a transition, because the 1-state of the cell's a-bit blocks all transitions of the neighbouring cells, thus preventing concurrent access to the state bits. Thus as long as a cell's a-bit is 1, the state bits of the cell may be updated in any order. This algorithm may not result in the most efficient implementations as it requires each cell to continuously check the a-bits of its neighbours. An alternative would be to implement a cell's complete logic functionality—including the above mutual exclusion algorithm—in terms of a delay-insensitive circuit that runs in without-busy-waiting mode. This would activate a cell only when strictly necessary. Finally, the above algorithm is not necessary if the cellular array can be reformulated such that two neighbouring cells are allowed to undergo simultaneous transitions, as in [70]. This would probably result in more cell states and transition rules, however, if implemented on the STCA. It tends to be more difficult to design such models and verify the correctness of their operation.

7. Conclusions and discussion

Asynchronous cellular arrays offer many advantages over synchronous computer architectures for implementations on nanometre scales. Their regular structure holds the promise for efficient manufacturing techniques based on directed molecular self-assembly. Their asynchronous mode of timing offers many advantages at high integration densities and is more compatible with the asynchronous nature of phenomena at nanometre scales. Consequently, physical implementations of asynchronous cellular arrays may be more straightforward than those of conventional computer architectures. Due to the nondeterminism accompanying their asynchronous nature, however, asynchronous cellular arrays are more difficult to compute on than synchronous architectures. The method developed in this paper makes this problem more manageable. The first element of this method, laying out a delay-insensitive circuit on an asynchronous cellular array, allows a cell to be passive as long as no signal is available for it. The second element of this method, partitioning the state of each cell into bits that are shared with the cell's neighbours, provides a strict control of the order in which cells undergo their transitions when they process a signal. Our method not only limits the required number of cell states and transition rules, but also allows the efficient exploitation of the cellular array's massive parallelism. Notwithstanding the model's simplicity, it is able to carry out the same class of computations as conventional computers. This paper aims to convey three messages.

(1) Cellular array-based architectures may be a better choice for nanocomputers than von Neumann computer architectures.
(2) An asynchronous mode of timing has advantages over a synchronous one for nanometre-scale implementations, and it is also manageable from a computational point of view.
(3) Rather than focusing on the realization of transistors on nanometre scales, it may pay off to find out how to implement and use operations that are typical for cellular arrays or delay-insensitive circuits.

There is a large body of work in delay-insensitive circuits that can be used with our approach. Previously proposed delay-insensitive circuits assume that at most one signal at a time can be on each wire, making them a special case of our circuits. To implement such delay-insensitive circuits on the asynchronous cellular array in this paper, we only need to express them in terms of the primitives in box 1. As pointed out in sections 3 and 5, systems as complicated as computers have been realized with delay-insensitive circuits, and much of this work, including design, implementation and debugging methods, can be carried over to our framework, in principle. The delay-insensitive circuits used in this paper, however, offer better efficiency for cellular automation-based implementations, though less experience is available with them. Their ability to transmit multiple signals at a time on each wire tends to decrease the need for feedback connections to acknowledge the receipt of signals. It also opens the way for the merge primitive used in this paper, which can accept signals at its inputs simultaneously—a precondition for the construction of without-busy-waiting sequencer modules from arbitrating test and set primitives (see section 3 and [52]).

In any distributed system there is a risk of deadlock, a state of a system in which different subparts are indefinitely waiting for input from each other, and the asynchronous systems in this paper are no exception to this. For example, two symmetric join primitives may mutually wait for input from each other, a situation that can never be resolved. Deadlock of this type is hard to prevent because it is due to incorrect circuit design. Provided circuits are correctly designed and laid out on the asynchronous cellular array, however, deadlock will not occur, because an undefined combination of state bits will never arise in any cell of such a cellular array.

In the introduction we briefly mentioned an alternative method to tame the asynchronicity of asynchronous cellular arrays, that is, by simulating synchronous cellular arrays on them [33, 35–37]. This approach requires more cell states and transition rules than ours: simulating an n-state synchronous cellular automaton requires a $3n^2$-state asynchronous cellular automaton in [33, 36], an $n^2 + 2n$-state asynchronous cellular automaton in [37], and an $O(n\sqrt{n})$-state generalized STCA in [35]. Worse yet, to synchronize different parts of the asynchronous cellular array with each other—necessary, because in the end it is a synchronous cellular array that is being simulated—exchange of signals between these parts is necessary. This exchange takes the form of waves—we call them *synchronization waves*—that propagate along the cellular space [36, 37]. Since all cells need to continuously change their states to accommodate these waves, there are a lot of dummy transitions, even in areas of the cellular array where no signals or configurations are present. Implemented physically, such an asynchronous cellular array needs to consume much more power, and with it dissipate heat, than should be strictly necessary to support its computation, which hardly makes it a better candidate for implementations in nanotechnology than synchronous cellular arrays.

The cells in a cellular array need to be as simple as possible to allow efficient implementations and manufacturing on nanometre scales. Assuming that the transition rules are stored locally in each cell, we count the complexity of a cell as the number of bits required to encode the cell's state and the table of transition rules. For the asynchronous cellular automaton in this paper this comes to four bits for a cell's state, eight bits for the left-hand side of each of the nine transition rules in box 2 and eight bits for the right-hand side, making a total of $4 + 9 \times (8 + 8) = 148$ bits. A reduction in the number of bits is possible by reducing the number of transition rules, and this may be accomplished if alternative delay-insensitive primitives are found that are simpler and fewer in number, yet form a universal set. An example of such an approach is in [74], in which the number of rules is reduced to four, resulting in only 68 bits per cell, but this is at the cost of efficiency. Though the same class of computations can be conducted on this model as on conventional computers, it is extremely limited in its ability: only a single signal at a time is allowed to run around in a delay-insensitive circuit mapped on this cellular array, hardly a model for efficient parallel information processing.

Our approach lends itself not only to the STCA cellular array in this paper, but also to more conventional asynchronous cellular arrays, i.e., models in which cells' states are treated as a whole, rather than being subdivided into substates (bits), and in which each cell can only update its own state, and not that of its neighbours. We have experimented with a five-state asynchronous cellular array with a von Neumann neighbourhood [70] and a six-state totalistic asynchronous cellular array with a Moore neighbourhood [75]. These cellular arrays require more transition rules, for example the five-state model in [70] requires 58 rules and the six-state model in [75] even more. For the model in [70], three bits are required to represent its five states. Moreover, the von Neumann neighbourhood requires each transition rule to encode in its left-hand side the states of a cell itself and of the four neighbour cells, and in its right-hand side the new state of the cell. This gives a total of $3+58\times3\times(1+4+1) = 957$ bits for encoding a cell's state and the table of transition rules. As this is substantially more than the number of bits required by our STCA model, this model may be more difficult to implement on nanometre scales than our model.

Are there ways to reduce the complexities of cells other than minimizing the number of transition rules and states? One standard way is to store the transition rules in a central place from which each cell can read them, but this does not work in our case. Not only does this require a communication structure for global access by all cells to the table of transition rules, a possibility that we try to avoid by using cellular arrays in the first place, it also requires each cell to continuously check whether a transition rule applies to it, which will swamp the central table by read requests from all cells and in the process consume much energy to support each cell's activities. The encoding of the transition table in each individual cell, on the other hand, allows implementations in which a cell becomes only active if a transition rule applies to it—a *data-driven* mode of operation that is intimately related to asynchronous systems.

Another strategy towards cells with low complexities is by exploiting the physical interactions between molecules to implement transition rules. The *cascades* of hopping CO molecules on a Cu(111) surface at cryogenic temperatures, recently presented by IBM researchers [51], are an example of such an approach. These molecules are arranged in configurations such that the motion of one molecule causes the subsequent motion of another, and so on in a cascade of motion similar to a row of toppling dominoes [51], which is a typical delay-insensitive mode of operation. The interactions of the CO molecules are used to realize the transmission of a signal along a path, as well as the operation of delay-insensitive primitives like a fork, a join and a merge, the latter three denoted in [51] as fan-out, AND-gate and OR-gate. A NOT-gate is implemented with dual-rail encoding by simple crossing the 0-wire and 1-wire. To implement a dual-rail encoded AND-gate in this framework, a symmetric join will be required [52], or a TRIA, or a resettable join. These are harder to implement by molecular cascades than a join—but probably not impossible. This scheme only provides *one-time computation*, as the original positions of the molecules are not recovered after an operation. To realize recoverable computations, it may be necessary to use molecules that hop between bi-stable states.

Quantum-dot cellular automata (QCAs) [21] is another model that exploits physical interactions as a way to implement transitions. This model comes usually in the form of a two-dimensional array of cells, each containing four quantum dots arranged near the corners of the cell, sometimes augmented with a fifth dot in the centre. In the ideal case the dots in a cell are occupied by two electrons, which will move to two dots in a cell's opposite corners due to electric (Coulomb) interactions, but which will never move out of the cell. Magnetic QCAs have also been proposed [76], and they seem to have great promise for operation at room temperature. The two different ways in which the electrons can settle in a cell are usually associated with a 0 and a 1, respectively. Lining up the cells in certain configurations gives rise to wires, fan-outs, AND-gates, NOT-gates etc. As it is hard to guarantee the arrival of a signal in a certain time interval in these models due to the complicated dynamics involved in the system converging to its ground state, a delay-insensitive mode of signalling seems suitable to them, though, on the other hand, it may be incompatible with the bounded decoherence times associated with QCA models. Clocked versions of QCAs have also been proposed [77]. As to the implementability of delay-insensitive circuits on QCAs, it is probably possible to implement a fork and a merge, because they resemble a fan-out and an OR-gate, respectively. A join may be more difficult, however, as it requires keeping an input signal pending in the case where the join waits for a second input signal to arrive.

Monomolecular arrays are promising for implementing STCA on nanometre scales. In such arrays, each cell consists of a single molecule that integrates the elementary functions and interconnections required for executing the cell's transitions. Currently, basic functions of digital electronics—rectification, amplification and storage—can be realized in single molecules, but connecting those molecules to one another poses difficulties, especially when it has to be done cost-effectively on a mass scale at high densities. Monomolecular electronics has the potential to solve this problem by integrating whole circuits within single molecules [78]. In recent years substantial progress has been made in the synthesis of very long molecules, and eventually it may be possible to realize a 15×15 nm^2 molecular circuit with as many as 1000 intramolecular switches [79], not a very large number to realize a computer, but sufficient to build a single cell. For this approach to be successful, a deeper understanding is required of the design of intramolecular electronics, which does not obey the known Kirchhoff law of electrical circuits [78, 79].

Is the asynchronous cellular array in this paper scalable? Though in principle the array can be made as large as we like, at certain sizes it will take a long time for a signal to move from one side of the array to the other side. For example, in a two-dimensional array of $10^9 \times 10^9$ cells each of which is able to do a transition every nanosecond on average, it takes a few seconds for a signal to traverse the array, suggesting that a two-dimensional array bigger than this may be impractical. The limits for three-dimensional arrays are less strict, but eventually they are also bounded in size. The issue of scalability is strongly connected to the organization of software on the array. If software is organized with an emphasis on locality, the need to traverse the array will be infrequent, making the above limitation be felt less strongly. Another factor that limits scalability is the input and output bandwidth of the cellular array. If input to and output from the array is conducted via the sides of the array, the upper bound on the bandwidth grows as the square root of the total number of cells in the case of a two-dimensional array and the cubic root in the case of a three-dimensional array. It is probably hard to circumvent this limitation other than organizing a system such that input and output is minimized. One way to do this is to use the array not only for computation, but also for permanent storage of programs and data, as envisioned in [16].

Though this paper shows how to efficiently simulate delay-insensitive circuits on asynchronous cellular arrays, it leaves open the challenge of configuring the inherently homogeneous cellular hardware into particular delay-insensitive circuits [19]. This problem boils down to moving a certain pattern of information—a configuration of cells in appropriate states representing a circuit layout—to a certain location in the cellular array. Assuming that the information to configure the cellular array originates from the sides of the cellular array, we see two configuration methods, both using only local interactions between cells. The first method employs cells that have two modes: a *computation* mode and a *configuration* mode. Used for example in the cell matrix [19], this method requires cells, each with a memory sufficiently large to contain a wide variety of transition rules and the capacity to write a combinatorial function of the contents of its neighbouring cells' memories into its own memory. Configuration is then achieved by rewriting the memories from cell to cell until the destination cell is reached. The second method uses the collective behaviour of the cells to copy information from one part in the cellular array to the other. Conceptually close to self-reproduction, this task has been successfully studied in synchronous cellular arrays [28, 29, 43–46], and it appears feasible as well in their asynchronous counterparts. To include configuration functionality into a cellular array with either method, however, the cells in the array will become somewhat more complex, and further research will be necessary.

When manufacturing on molecular scales, it is inevitable that some of the cells contain defects, and, even if cells work properly, they will now and then err due to noise and quantum effects. To cope with this, a cellular array needs to be defect tolerant and fault tolerant. Defect tolerance, the ability of a machine to work properly notwithstanding persistent defects, is essential for creating working nanocomputers. A cellular array would ideally detect its own defects and cope with them, rather than leaving this to an outside computer [20], which is likely to be a bottleneck. Fault tolerance, the ability to correct nonpersistent errors, has been studied extensively in conventional computers, and has also attracted attention in the context of cellular arrays [34, 80, 81]. In principle defect tolerance and fault tolerance are possible in asynchronous cellular arrays, but further research is required to design models implementing these concepts without the complexities of the cells increasing too much.

Future research on nanocomputers based on asynchronous cellular arrays needs to focus on the items mentioned in this discussion, in particular on decreasing the complexity of cells, physical implementations, configuration of asynchronous cellular arrays and fault and defect tolerance. Another topic of interest when building nanocomputers is *reversible computing* [82], a way of computing that is backwards deterministic. Every operation in a reversible computation is governed by a one-to-one function, which implies that information never gets destroyed. As with asynchronous computing, reversibility tends to reduce power consumption and heat dissipation: in principle, a reversible computation can be conducted without consuming energy, whereas each irreversible operation conducted at temperature T requires energy of at least $kT \ln 2$ J/bit [83]. Though studied extensively in a great variety of synchronous computation models, reversible computing is virtually unexplored in an asynchronous framework. As most physical interactions on the nanometre scale are asynchronous, it makes sense to investigate whether a reversible mode of computation can be included in an asynchronous framework. Some preliminary results on the combination of reversibility and delay insensitivity are in [84], which proposes delay-insensitive circuits that, while not strictly reversible, have the flavour of it. Lee *et al* [74] formulate a delay-insensitive reversible implementation on an STCA cellular array, but the resulting model is inefficient, since it only allows a single signal to run around at a time in the whole cellular array.

Acknowledgments

We thank Kenichi Morita at Hiroshima University, Satoru Miyauchi at the Communications Research Laboratory in Kobe and Nobuyuki Matsui and Teijiro Isokawa at Himeji Institute of Technology (all institutes in Japan) for their support and discussions. We also thank the two anonymous reviewers for their extensive and valuable comments and suggestions.

References

[1] Bachtold A, Hadley P, Nakanishi T and Dekker C 2001 Logic circuits with carbon nanotube transistors *Science* **294** 1317–20

[2] Chen J, Reed M A, Rawlett A M and Tour J M 1999 Observations of a large on–off ratio and negative differential resistance in an electronic molecular switch *Science* **286** 1550–2

[3] Chen J *et al* 2000 Room-temperature negative differential resistance in nanoscale molecular junctions *Appl. Phys. Lett.* **77** 1224–6

[4] Collier C P *et al* 1999 Electronically configurable molecular-based logic gates *Science* **285** 391–4

[5] Cui Y and Lieber C 2001 Functional nanoscale electronic devices assembled using silicon nanowire building blocks *Science* **291** 851–3

[6] Derycke V, Martel R, Appenzeller J and Avouris Ph 2001 Carbon nanotube inter- and intramolecular logic gates *Nano Lett.* August

[7] Ellenbogen J C and Love J C 1999 Architectures for molecular electronic computers: I. Logic structures and an adder built from molecular electronic diodes *MITRE Corporation Reports* available at http://www.mitre.org/technology/nanotech
Ellenbogen J C 1999 Architectures for molecular electronic computers: II. Logic structures using molecular electronic FETs *MITRE Corporation Reports* available at http://www.mitre.org/technology/nanotech

[8] Huang Y *et al* 2001 Logic gates and computation from assembled nanowire building blocks *Science* **294** 1313–17

[9] Postma H W Ch, Teepen T, Yao Z, Grifoni M and Dekker C 2001 Carbon nanotube single-electron transistors at room temperature *Science* **293** 76–9

[10] Reed M A, Chen J, Rawlett A M, Price D W and Tour J M 2001 Molecular random access memory cell *Appl. Phys. Lett.* **78** 3735–7

[11] Rueckes T *et al* 2000 Carbon nanotube-based nonvolatile random access memory for molecular computing *Science* **289** 94–7

[12] Wada Y, Uda T, Lutwyche M, Kondo S and Heike S 1993 A proposal of nano-scale devices based on atom/molecule switching *J. Appl. Phys.* **74** 7321–8

[13] Wada Y 1996 Atom electronics *Microelectron. Eng.* **30** 375–82

[14] Porod W 2002 Nanoelectronic circuit architectures *Handbook of Nanoscience, Engineering, and Technology* ed W A Goddard III, D W Brenner, S E Lyshevski and G J Lafrate (Boca Raton, FL: Chemical Rubber Company Press) ch 5

[15] Compañó R (ed) 2001 *Technology Roadmap for Nanoelectronics* (Luxembourg: Office for Official Publications of the European Communities)

[16] Beckett P and Jennings A 2002 Towards nanocomputer architecture *Proc. 7th Asia–Pacific Computer Systems Architecture Conf., ACSAC'2002 (Conf. on Research and Practice in Information Technology)* vol 6, ed F Lai and J Morris

[17] Biafore M 1995 Cellular automata for nanometre-scale computation *Physica* D **70** 415–33

[18] DeHon A 2002 Array-based architecture for molecular electronics *Proc. 1st Workshop on Non-Silicon Computation, NSC-1*

[19] Durbeck L J K and Macias N J 2001 The cell matrix: an architecture for nanocomputing *Nanotechnology* **12** 217–30

[20] Heath J R, Kuekes P J, Snider G S and Williams R S 1998 A defect-tolerant computer architecture: opportunities for nanotechnology *Science* **280** 1716–21

[21] Lent C S and Tougaw P D 1997 A device architecture for computing with quantum dots *Proc. IEEE* **85** 541–57

[22] Lyke J, Donohoe G and Karna S 2001 Reconfigurable cellular array architectures for molecular electronics *Air Force Research Laboratory Report* AFRL-VS-TR-2001-1039 available at http://www-2.cs.cmu.edu/~phoenix/internal/papers_by_others/TR-2001-1039.PDF

[23] Peper F 2000 Spatial computing on self-timed cellular automata *Proc. 2nd Conf. on Unconventional Models of Computation, UMC'2K* (Berlin: Springer) pp 202–14

[24] Seminario J M and Tour J M 1998 *Ab initio* methods for the study of molecular systems for nanometer technology: towards the first principles of molecular computers *Molecular Electronics: Science and Technology (Ann. NY Acad. Sci. vol 852)* ed A Aviram and M Ratner pp 68–94

[25] Fountain T J, Duff M J B, Crawley D G, Tomlinson C D and Moffat C D 1998 The use of nanoelectronic devices in highly parallel computing systems *IEEE Trans. Very Large Scale Integr. Syst.* **6** 31–8

[26] Waingold E *et al* 1997 Baring it all to software: raw machines *IEEE Comput.* **30** 86–93

[27] Kung S Y 1982 Why systolic architectures? *Computer* **15** 37–46

[28] Von Neumann J 1966 *Theory of Self-Reproducing Automata* ed A W Burks (Champaign, IL: University of Illinois Press)

[29] Codd E F 1968 *Cellular Automata* (New York: Academic)

[30] Wolfram S 1994 *Cellular Automata and Complexity* (Reading, MA: Addison-Wesley)

[31] Hopcroft J E, Motwani R and Ullman J D 2001 *Introduction to Automata Theory, Languages, and Computation* (Reading, MA: Addison-Wesley)

[32] Fredkin E and Toffoli T 1982 Conservative logic *Int. J. Theor. Phys.* **21** 129–253

[33] Nakamura K 1974 Asynchronous cellular automata and their computational ability *Syst. Comput.—Controls* **5** 58–66

[34] Wang W 1991 An asynchronous two-dimensional self-correcting cellular automaton *PhD Thesis* Boston University

[35] Peper F, Isokawa T, Kouda N and Matsui N 2002 Self-timed cellular automata and their computational ability *Future Gener. Comput. Syst.* **18** 893–904

[36] Nehaniv C L 2002 Self-reproduction in asynchronous cellular automata *Proc. NASA/DoD Conf. on Evolvable Hardware, EH'02* pp 201–9

[37] Lee J, Adachi S, Peper F and Morita K 2003 Asynchronous game of life, in preparation

[38] Hauck S 1995 Asynchronous design methodologies: an overview *Proc. IEEE* **83** 69–93

[39] Davis A and Nowick S M 1997 An introduction to asynchronous circuit design *Technical Report* UUCS-97-013 Computer Science Department, University of Utah, Downloadable from http://www.cs.columbia.edu/async/publications.html

[40] Keller R M 1974 Towards a theory of universal speed-independent modules *IEEE Trans. Comput.* **C-23** 21–33

[41] Patra P 1995 Approaches to design of circuits for low-power computation *PhD Thesis* University of Texas at Austin

[42] Myers C J 2001 *Asynchronous Circuit Design* (New York: Wiley)

[43] Langton C G 1984 Self-reproduction in cellular automata *Physica* D **10** 135–44

[44] Reggia J A, Armentrout S L, Chou H-H and Peng Y 1993 Simple systems that exhibit self-directed replication *Science* **259** 1282–7

[45] Morita K and Imai K 1996 Self-reproduction in a reversible cellular space *Theor. Comput. Sci.* **168** 337–66

[46] Serizawa T 1987 Three-state Neumann neighbour cellular automata capable of constructing self-reproducing machines *Syst. Comput. Japan* **18** 33–40

[47] Petraglio E, Tempesti G and Henry J-M 2002 Arithmetic operations with self-replicating loops *Collision-Based Computing* ed A Adamatzky (Berlin: Springer) pp 469–90

[48] Adamatzky A (ed) 2002 *Collision-Based Computing* (Berlin: Springer)

[49] 2001 It's time for clockless chips *Technol. Rev. Mag.* **104** 36–41 available at: http://www.cs.columbia.edu/~nowick/async-intro.html

[50] Matzke D 1997 Will physical scalability sabotage performance gains? *IEEE Comput.* **30** 37–9

[51] Heinrich A J, Lutz C P, Gupta J A and Eigler D M 2002 Molecular cascades *Science* **298** 1381–7

[52] Lee J, Peper F, Adachi S and Morita K 2002 Compact designs of universal delay-insensitive circuits with bi-directional and buffering lines in preparation

[53] Unger S H 1969 *Asynchronous Sequential Switching Circuits* (New York: Wiley)

[54] Seitz C L 1980 System timing *Introduction to VLSI Systems* ed C Mead and L Conway (Reading, MA: Addison-Wesley) ch 7

[55] Muller D E and Bartky W S 1959 A theory of asynchronous circuits *Proc. Int. Symp. on the Theory of Switching* (Cambridge, MA: Harvard University Press) pp 204–43

[56] Martin A J 1990 Programming in VLSI: from communicating processes to delay-insensitive circuits *Developments in Concurrency and Communication (UT Year of Programming Institute on Concurrent Programming)* ed C A R Hoare (Reading, MA: Addison-Wesley) pp 1–64

[57] Martin A J 1990 The limitations to delay-insensitivity in asynchronous circuits *Proc. 6th MIT Conf. on Advanced Research in VLSI* (Cambridge, MA: MIT Press) pp 263–78

[58] van Berkel K 1992 Beware the isochronic fork *Integr. VLSI J.* **13** 103–28

[59] Clark W A 1967 Macromodular computer systems *Proc. Conf. Spring Joint Computer Conf. (AFIPS)* pp 335–6

[60] Ornstein S M, Stucki M J and Clark W A 1967 A functional description of macromodules *Proc. Conf. Spring Joint Computer Conf. (AFIPS)* pp 337–55

[61] Molnar C E, Fang T-P and Rosenberger F U 1985 Synthesis of delay-insensitive modules *Chapel Hill Conf. on Very Large Scale Integration* ed H Fuchs (Rockville, MD: Computer Science Press) pp 67–86

[62] Rosenberger F U, Molnar C E, Chaney T J and Fang T-P 1988 Q-modules: internally clocked delay-insensitive modules *IEEE Trans. Comput.* **37** 1005–18

[63] Ebergen J C 1991 A formal approach to designing delay-insensitive circuits *Distrib. Comput.* **5** 107–19

[64] Unger S H 1993 A building block approach to unclocked systems *Proc. Hawaii Int. Conf. on System Sciences* vol 1 (Los Alamitos, CA: IEEE Computer Society Press) pp 339–48

[65] Sutherland I E 1989 Micropipelines *Commun. ACM* **32** 720–38

[66] Muller D E 1963 Asynchronous logics and application to information processing *Switching Theory in Space Technology* (Stanford, CA: Stanford University Press)

[67] Brzozowski J A and Ebergen J C 1992 On the delay-sensitivity of gate networks *IEEE Trans. Comput.* **41** 1349–60

[68] Ebergen J C 1987 Translating programs into delay-insensitive circuits *PhD Thesis* Technical University Eindhoven

[69] *Encyclopedia of Delay-Insensitive Systems (EDIS)* http://edis.win.tue.nl/edis.html

[70] Lee J, Adachi S, Peper F and Morita K 2002 Embedding universal delay-insensitive circuits in asynchronous cellular spaces, in preparation

[71] Movie of signals crossing each other (referred to in Box 3) is available at: http://www.iop.org/EJ/S/2/IOPP/mmedia/0957-4484/14/4/312

[72] Movie (referred to in Box 4) of writing value 0 in one-bit memory, writing value 1 and reading out the value of the memory, respectively, is available at: http://www.iop.org/EJ/S/2/IOPP/mmedia/0957-4484/14/4/312

[73] van Berkel C H and Saeijs R W J J 1988 Compilation of communicating processes into delay-insensitive circuits *Proc. IEEE Int. Conf. on Computer Design* (Los Alamitos, CA: IEEE Computer Society Press) pp 157–62

[74] Lee J, Peper F, Adachi S, Morita K and Mashiko S 2002 Reversible computation in asynchronous cellular automata *Proc. 3rd Conf. on Unconventional Models of Computation, UMC'02* (Berlin: Springer) pp 220–9

[75] Adachi S, Peper F and Lee J 2002 Computation by asynchronously updating cellular automata, in preparation

[76] Cowburn R P and Welland M E 2000 Room temperature magnetic quantum cellular automata *Science* **287** 1466–8

[77] Orlov A *et al* 2000 Experimental demonstration of clocked single-electron switching in quantum-dot cellular automata *Appl. Phys. Lett.* **77** 295–7

[78] Joachim C, Gimzewski J K and Aviram A 2000 Electronics using hybrid-molecular and mono-molecular devices *Nature* **408** 541–8

[79] Joachim C 2002 Bonding more atoms together for a single molecule computer *Nanotechnology* **13** R1–7 (tutorial)

[80] Gács P 2001 Reliable cellular automata with self-organization *J. Stat. Phys.* **103** 45–267

[81] Macias N J and Durbeck L J K 2002 Self-assembling circuits with autonomous fault handling *Proc. NASA/DoD Conf. on Evolvable Hardware, EH'02* pp 46–55

[82] Bennett C H 1988 Notes on the history of reversible computation *IBM J. Res. Dev.* **32** 16–23

[83] Landauer R 1961 Irreversibility and heat generation in the computing process *IBM J. Res. Dev.* **5** 183–91

[84] Patra P and Fussell D S 1996 A framework for conservative and delay-insensitive computing *University of Texas at Austin Technical Report* TR-95-10