Fault-Tolerance in Nanocomputers: A Cellular Array Approach

Ferdinand Peper, *Member, IEEE*, Jia Lee, Fukutaro Abo, Teijiro Isokawa, *Member, IEEE*, Susumu Adachi, Nobuyuki Matsui, and Shinro Mashiko

***Abstract*—Asynchronous cellular arrays have gained attention as promising architectures for nanocomputers, because of their lack of a clock, which facilitates low power designs, and their regular structure, which potentially allows manufacturing techniques based on molecular self-organization. With the increase in integration density comes a decrease in the reliability of the components from which computers are built, and implementations based on cellular arrays are no exception to this. This paper advances asynchronous cellular arrays that are tolerant to transient errors in up to one third of the information stored by its cells. The cellular arrays require six rules to describe the interactions between the cells, implying less complexity of the cells as compared to a previously proposed (nonfault-tolerant) asynchronous cellular array that employs nine rules.**

***Index Terms*—Cellular array, computer architecture, error correction, fault-tolerance, homogeneous structure, nanocomputing.**

I. Introduction

THE successful construction and manipulation of devices on nanometer scales ([1]–[7]; short review in [8]) has resulted in increasing interest in building computers from them, even though the technology is still far from the stage of mass-production. In the debate about the architecture most suitable for such nanocomputers, there is a growing consensus that a regular structure of locally connected elements will facilitate efficient physical implementations and manufacturing methods in which parts are put together in a self-organized way using chemical synthesis ([9]–[14]).

Promising in this respect are *Cellular Arrays*. Consisting of a large number of simple identical cells organized as two-(2-D) or three-dimensional (3-D) arrays, they have been widely studied as models of computation. Interactions between cells are modeled by a small number of transition rules that act on a local level, involving only the direct neighbors of a cell and the cell itself. Most cellular array models assume a global timing scheme according to which all cells undergo transitions at each time step. Alternative models, called *asynchronous* cellular arrays [13], [15]–[18], assume a timing model according to which the state transitions of each cell take place at random times, independently of other cells. An asynchronous mode of timing may benefit physical implementations of cellular arrays in the same way it benefits asynchronous circuits (e.g., [19]): the absence of a central clock eliminates the need to distribute its signals to all elements, and it may also result in faster circuits consuming less power and, related to the latter, dissipating less heat. These advantages may be especially important with alternative technologies, like molecular electronics [20], [21], RSFQ superconducting technology [22], [23], quantum dot cellular automata [24], [25], etc. Efficient schemes to compute on asynchronous cellular arrays [13], [16]–[18] are based on the simulation of *delay-insensitive circuits*, a type of asynchronous circuit of which the correctness of operation is robust to delays of signals (e.g., see [19]). In this approach, configurations of cells are designed such that they implement a set of primitive circuit elements from which any arbitrary delay-insensitive circuit can be built [13]–[18] (see also [26]).

Robustness to errors is an important design consideration for nanocomputers in the light of the noise and instabilities that affect the reliability of nanometer-scale devices. Heath *et al.* [27] distinguish two types of robustness to errors: *Fault Tolerance*, the ability to recover from transient errors during computations—the main topic of this paper—and *Defect Tolerance*, the ability to operate flawless in the presence of permanent hardware errors that emerged in the manufacturing process—a topic only briefly discussed in the last section of this paper. The study of reliable computation by unreliable elements originates with von Neumann [28]. In his *Multiplexing* technique, each wire in a circuit is replaced by a bundle of wires on which a majority vote is conducted to establish its value. Dobrushin and Ortyukov, formalizing this technique, show in [29] that a function computed by m fault-free elements can be reliably computed by $\mathrm{O}(m \log m)$ unreliable elements with a high probability. This is further improved in [30], [31] to $\mathrm{O}(m)$ unreliable elements. Another method, *R-fold modular redundancy* [32], achieves tolerance to faults by employing R copies of a unit (R preferably an odd number), of which the majority establishes the most likely output the unit would have were it flawless, in accordance with an operation conducted by a *majority gate*. If $R = 3$, this technique is called *triple modular redundancy* (TMR). Combining the outputs of three TMR units by a majority gate on a second level and so on in a hierarchy of levels, we obtain *Cascaded Triple Modular Redundancy* (CTMR) [33], a model with increased reliability higher in the hierarchy. In the context of nanotechnology, the merits of TMR, CTMR, and von Neumann's multiplexing technique have been found to be limited [34]: to make a chip with 10^{12} devices work with a probability of 90%, an unrealistically small device fault rate for nanometer-scale implementations and a large hardware redundancy is required. For example, when employing the multiplexing technique, an

Manuscript received August 1, 2003; revised October 17, 2003.

F. Peper, J. Lee, S. Adachi, and S. Mashiko are with the Communications Research Laboratory, Nanotechnology Group, Kobe 651-2492, Japan.

F. Abo, T. Isokawa, and N. Matsui are with Himeji Institute of Technology, Division of Computer Engineering, Hyogo 671-2201, Japan.

Digital Object Identifier 10.1109/TNANO.2004.824034

error rate of 10^{-3} per device requires a redundancy of almost a factor 10^5. Such a high redundancy would down grade the chip to one effectively having only 10^7 devices, which is hardly an improvement over current technology. For the TMR technique, the situation is even worse. In [35] an improvement to a redundancy factor from 100 to 1000 is given. These results are further improved upon in [36], whereby von Neumann's multiplexing technique is applied in a hierarchical way in combination with reconfiguration techniques. The claim is that a redundancy factor of 10 would suffice in this design, even with a device error rate of up to 0.01.

The fault-tolerant construction of von Neumann and its follow-up by others is basically an *error correcting code* (e.g., [37]). Error correcting codes provide a way to cope with the corruption of bits by encoding messages as codewords that contain redundant information. This information is used to reconstruct the original codeword in case errors occur. Error correcting codes are a more systematic way to deal with errors, as compared to methods like TMR, and they tend to require less redundancy. Von Neumann's construction boils down to the use of a so-called *repetition code*, a code (but not a very efficient one at that) in which each symbol of a message is repeated many times to create redundancy. Computation in this scheme can then be considered as taking place in the encoded space, whereby errors are corrected locally, and encoding and decoding is only necessary at the beginning and the end, respectively, of the computation. This line of thought is also followed in [38], but then with *generalized Reed–Solomon codes* (e.g., [37]) used, rather than repetition codes, to realize a fault-tolerant computing model with improved reliability.

In the context of cellular arrays, early work on fault-tolerance is reported in [39]. This model can correct at most one error in 19 cells, but to this end each cell needs read access to the states of 49 cells in its neighborhood, which is much higher than the four cells usual in cellular arrays (von Neumann neighborhood). The increased complexity of cells suggests that they will be very error-prone in physical implementations, and thus that this work is mainly of theoretical value. The model in [40] suffers from similar problems. Better fault-tolerance is obtained in [41]–[43] with synchronous cellular arrays, and in [44], [45] with asynchronous cellular arrays simulating synchronous cellular arrays: the idea is to organize cells in blocks that perform a fault-tolerant simulation of a second cellular array, which on its turn is also organized in blocks, simulating even more reliably a third cellular array, and so on. This results in a hierarchical structure with high reliability at the higher levels, like with the CTMR technique. The cells in these models are too complicated to be suitable for physical implementations, however, since they contain information regarding the hierarchical organization, such as block structure, address within a block, programs selecting transition rules, and timing information. In [46] a fault-tolerant asynchronous cellular array based on *Bose–Chaudhuri–Hocquenghen (BCH) codes* (e.g., [37]) is proposed, but computation on this model is inefficient, since only one signal at a time is allowed.

This paper contains two main results. First, an asynchronous cellular array is proposed that conducts computations efficiently by simulating delay-insensitive circuits, whereby only six transition rules are required to describe the functionality of the cells. This compares to nine transition rules required by the asynchronous cellular array in [13]. The substantial reduction in the number of transition rules is due to the use of a novel type of delay-insensitive circuit [47]. In the second part of this paper, we make this cellular array fault-tolerant to arbitrary errors in up to asymptotically one third of the bits representing the information in the cellular array. We do so by encoding the state information of each cell as codewords in an error correcting code, like in [46], and show that our scheme is superior to schemes employing a repetition code. Moreover, we achieve tolerance to many error patterns of which up to almost half of the bits is corrupted. Finally, it is shown that the mechanism for conducting transitions in each cell can be divided into small units, each of which operates independently on a part of the bits representing the state information of a cell. Errors caused by faults in part of the units can be corrected by the proposed scheme.

This paper is organized as follows. Section II describes the asynchronous cellular array model used, and Section III gives a short explanation of delay-insensitive circuits. Section IV describes the construction of the asynchronous cellular array based on six transition rules, and it shows how a delay-insensitive NAND-gate and 1-bit memory can be implemented on the cellular array. Fault-tolerant versions of the cellular array are given in Section V. Section VI describes how the mechanism for state transitions in a cell can be divided into independent units, the output of which is operated upon by the error correcting mechanism. We finish with conclusions and a discussion.

II. Asynchronous Cellular Arrays

A cellular array is a 2-D array of identical cells, each of which has four neighboring cells, at its north, its east, its south, and its west. A 3-D version is also possible, but is not considered here. Each side of a cell has a memory of a few bits attached to it,[1] as in Fig. 1. The four memories at the side of a cell are said to be *associated* with the cell. The *state* of a memory is the value stored in it. The assignment of a particular combination of states of the memories associated with a certain set of cells in the cellular automaton is called a *configuration*. A cell may change the states of the four memories associated with it according to an operation called a *transition*. The transitions a cell may undergo are expressed by a table of *transition rules*. A transition rule applies to a cell if its left-hand side matches the combination of states of the memories associated with the cell. The right hand side of the rule describes into what states the memories are changed if the rule is applied (Fig. 2). As a memory is shared between two cells, it may be updated by a transition acting on either of these cells. Certain desired behavior of a cellular array can be obtained by setting its memories in proper states and defining transition rules that lead to state changes corresponding to this behavior. For example, one may design a cellular array that can conduct the same class of computations as those possible on a conventional computer [13].

Cellular arrays in which all cells undergo transitions at the same time are called *synchronous*. They are the most widely

[1]Though cellular arrays may have different forms [48], [49], we only consider the type in this paper, because it requires few transition rules, which is a prerequisite for simple cells. The cellular arrays in this paper resemble so-called *partitioned cellular automata* [50].

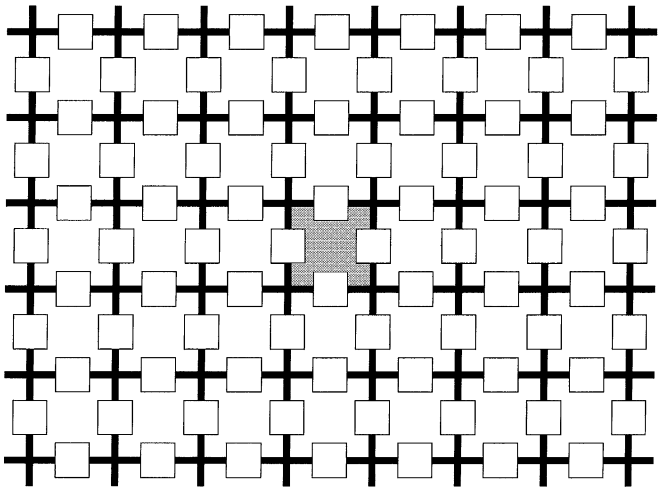


Fig. 1. Cellular array consisting of cells (the big squares with fat borders, one of which is shaded), each having access to four memories (the small squares). Shared by two cells, a memory stores a few bits of information. The cells operate on these memories with a small number (six in this paper) of transition rules.

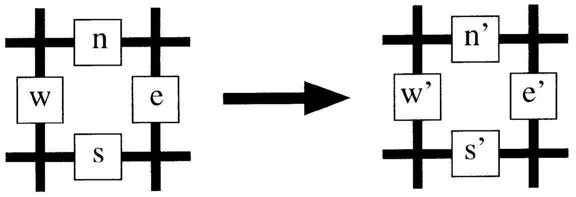


Fig. 2. Transition rule describing a transition the memories associated with a cell can undergo. If the memory states of a cell match the states n, e, s, and w in the left hand side of the rule, the rule may be applied to the cell, as a result of which the states are changed into the states n', e', s', and w', respectively, depicted on the right-hand side.

studied type. When transitions of the cells occur at random times, independent from each other, we obtain *asynchronous cellular arrays*, the type employed in this paper. In an asynchronous cellular array, a cell may undergo a transition when its memory states match the left hand side of a transition rule, and this operation is randomly timed. If there is no transition rule whose left hand side matches a cell's combination of memory states, the cell remains unchanged. Though transitions of cells are timed randomly and independently of each other, they are subject to the condition that two neighboring cells never undergo transitions simultaneously. This ensures that two neighboring cells will not attempt to set the memory shared between them to different states at the same time—a write conflict that could lead to an undefined state of the memory. Possible mechanisms to avoid the simultaneous update of two neighboring cells can be found in [13].[2]

The asynchronous cellular array in the first part of this paper has memories of two bits each, a model also known under the name *self-timed cellular automaton (STCA)* [13], [53]. Interconnection lines in this model take the form of straight continuous areas of cells, called *paths*. When there are no signals on a path, its cells have all their memories in state 0, that is, all bits in each of the memories are 0. A signal on a path is represented as a cell with a combination of memory states, in which three of the cell's memories are in state 0, and the remaining memory contains one 1-bit and one 0-bit. Fig. 3(a) shows an example of a transition rule that, operating on a signal, moves the signal's 1-bit toward the memory one cell in the northern direction. Applied twice, the transition rule gives rise to the sequence of configurations in Fig. 3(b) in which the 1-bit propagates in the northern direction along a path 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 toward the south, east, or west as well.

We have thus defined a model in which each cell performs transitions on information stored in the memories associated with it. Later on, we equip the memories with error correcting ability by adding bits to them that store redundancy information.

III. 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 lines. 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 signals, and becomes inactive again. Not requiring a clock, delay-insensitive circuits belong to the larger class of *asynchronous circuits*, which have several advantages over synchronous circuits [19], [54]. In the context of nanotechnology implementations we underline the following advantages (see also [13]):

1) Distributing the clock signal is not needed. This not only saves on circuit area, but also facilitates the homogeneity of the system—an important issue for manufacturing based on molecular self-assembly. Moreover, many timing problems disappear, like the failure of signals to reach their destination within a single clock cycle.
2) Energy consumption and heat dissipation are reduced, because only those parts of the circuit in which signals reside need to be active.
3) Changes in the timings of signals, caused by variations in physical conditions or implementations, do not affect the correct operation of asynchous circuits, especially if they are delay-insensitive.
4) Asynchronous circuits can be divided into modules that are designed without considering other modules, especially with respect to timing relationships. Modularity facilitates *configurability* of circuits.

The above advantages are felt even stronger at higher integration densities, though for some technologies, such as CMOS, it is hard to exploit them, leading to performances that are actually worse with respect to power consumption, wiring requirements, and speed. Alternative technologies, like molecular electronics and RSFQ superconducting technology, may suffer less from such problems. For a more extensive discussion of

[2]This scheme can be interpreted as an asynchronous version of a checkerboard updating scheme based on the Margolus neighborhood [51], [52], whereby the cellular array is subdivided in cells in an alternative way, i.e., by drawing diagonal lines through the corners of the cells and interpreting the resulting diagonal squares as cells. Each newly obtained cell then contains one of the original memories. The four memories associated with an original cell then correspond with a block of 2×2 cells, to be used as a Margolus neighborhood.

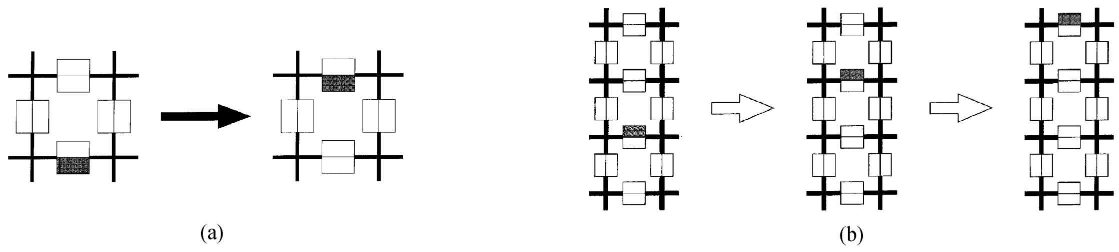


Fig. 3. (a) Transition rule for signal propagation, and (b) its application twice to a configuration of cells, each time giving rise to a signal successively moving one cell toward the north. Here a memory contains two bits, each of which is indicated by a block that is shaded for the value 1 and white for the value 0. The transition rule operates on all bits in the four memories associated with a cell.

asynchronous circuits in the context of nanotechnology we refer to [13].

Delay-insensitive circuits are built up of *modules* (see also [13], [47], [55]), which are elements with a finite number of input and output lines and a finite number of states. When receiving certain signals from its input lines, a module conducts an operation, as a result of which it may change its state and output certain signals on its output lines. Modules can be constructed from networks of simpler modules, and they may be as complex as a delay-insensitive computer. Realized in a cellular array by configurations of cells in certain states, modules are connected to each other by paths of cells.

A signal propagates from one end of a path (the *source*) to the other end (the *destination*), using the method lined out in Fig. 3 in the previous section. Once a signal is transmitted from a source, it cannot be withdrawn, and it will head for its destination. There can be more than one signal on each path, but signals never interfere with each other on a path, except that they may delay each other for a finite amount of time [47], [55].

When a 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*. If a module is presented with two input signals from different input paths, and it can only process one at a time, it may arbitrarily choose which signal to process first. It is then said to *arbitrate* between its input signals. The transmission of signals over paths and their processing by modules may undergo a finite time delay, without this having consequences for the correct operation of the circuit. Systematic overviews of conditions under which delay-insensitive circuits operate can be found in [18], [47], [55], [56].

As with Boolean circuits in a synchronous system, which can be constructed using only AND-gates and NOT-gates, it is possible to construct delay-insensitive circuits from a limited set of *primitive modules* or *primitives*. Such a set, called *universal*, can be used for designing circuits that conduct the same class of computations as possible on conventional computers (see also discussion in [55]). A universal set of primitives for delay-insensitive circuits, like the sets in [18], [56]–[58], provides both universal logic functionality as well as timing functionality, the latter being necessary to make up for the absence of a clock. A small number of input or output paths of primitives is useful for implementations on cellular arrays, as each cell has only a limited number of neighbors through which input and output may take place, a limitation strongly felt if the cellular array is 2-D. The universal set of primitives in [55] requires only three paths by each primitive module for input or output of signals. The key point of this set is the use of paths that are bidirectional and that can contain more than one signal at a time, provided they all move in the same direction. Due to the high number of primitives (five) it consists of, however, an implementation of this set on an asynchronous cellular array requires as many as nine transition rules [13]. In this paper we use an alternative set of delay-insensitive primitives recently proposed in [47]. This set can do without bi-directionality of paths, but it retains paths that may contain multiple signals at a time. Though the maximum number of paths for input or output of signals required by the primitives increases to four from three, the set contains only three primitives (see also [18] for a set of four primitives each with four paths). This results in an asynchronous cellular automaton implementation requiring only six transition rules, as we will see in Section IV.

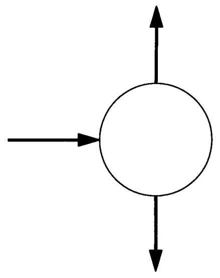
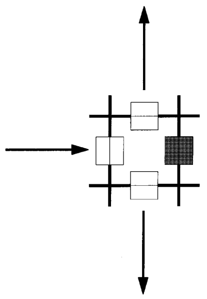
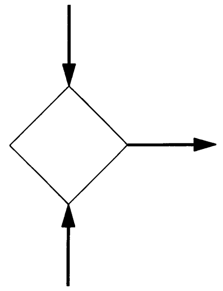
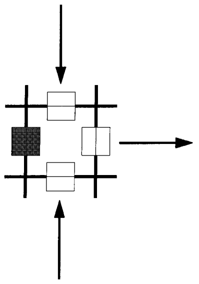
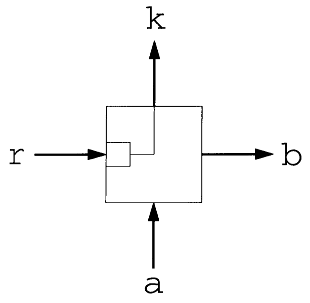
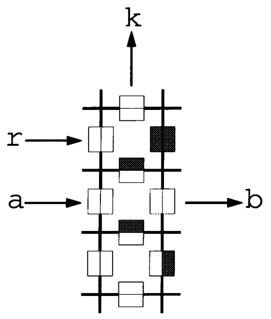
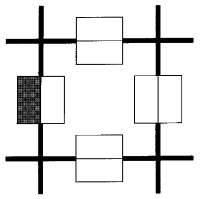
IV. Implementing Delay-Insensitive Circuits on Asynchronous Cellular Arrays

The set of delay-insensitive primitives used in this paper are listed in Table I, the left side denoting the symbols for the primitives, and the right side the cell configurations used to implement them on the cellular array. A delay-insensitive circuit can be realized on the cellular array by connecting configurations to each other by paths of cells such that the inputs and outputs of the primitives are appropriately lined up. It is shown in [47] that any arbitrary delay-insensitive circuit can be constructed in this way. The cell configurations behave like the primitives according to the set of six transition rules defined in Table II. Only when the configurations are presented with input signals do they undergo transitions, as the left hand sides of the transition rules fail to match them otherwise. The configurations are thus stable in the absence of input signals.

The *Fork* primitive is a fan-out element commonly used in delay-insensitive circuits [56], [59]. It produces one signal on each of its two output paths for every signal it receives from its input path in accordance with rule 2 in Table II (see Fig. 4).

The *Merge* primitive, proposed in [55], merges two streams of input signals into one stream of output signals. It is represented by the same configuration as the Fork, but then with input signals arriving from its sides, and it processes a single signal in accordance with rule 3 in Table II, as in Fig. 5(a). The Merge primitive also allows simultaneous input signals to its

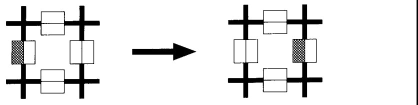
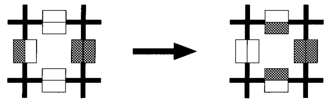
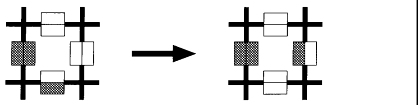
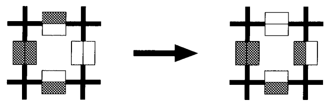
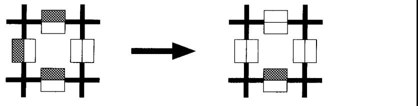
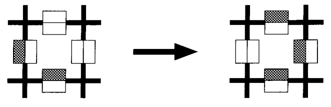
TABLE I
PRIMITIVES OF DELAY-INSENSITIVE CIRCUITS: SYMBOLS, FUNCTIONAL DESCRIPTIONS, AND IMPLEMENTATIONS ON THE CELLULAR ARRAY

Symbol	Functionality	Configuration
	**Fork**: A module 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.	
	**Merge**: A module with two input paths and one output path. Signals arriving from the input paths are redirected to the output path. Simultaneous input signals are allowed, giving rise to two consecutive output signals.	
	**Resettable Modulo 2 Counter (R-Counter)**: A module with two input paths, *a* and *r*, and two output paths, *b* and *k*. Two successive input signals on *a* give rise to one output signal on *b*. One input signal on *a* and one on *r* give rise to one output signal on *k*. If there are two inputs to *a* and one input to *r*, one of the above two possibilities will be arbitrarily chosen. If there is only one input to either *a* or *r*, no output will emerge until one more signal is input to *a*.	
	Signal: A signal propagating on a path (see also Fig. 3). Other signals on the same path may delay the signal, but never change it.	

two input paths, unlike previously proposed Merge primitives [56]–[58], [60], which require mutually exclusive inputs. Our Merge processes simultaneous inputs at its paths by first forwarding one signal to the output path, while keeping the input signal at the other input path pending in accordance with rule 4, as in Fig. 5(b). After this, it is ready to handle the other input, and forward it to the output path, using rule 3. This primitive can also be used for right and left turns of signals. For more details and background of this Merge module we refer to [13], since it has also been used therein.

The *Resettable Modulo 2 Counter (R-Counter)* primitive counts (modulo 2) the input signals to path a, producing one output signal on path b for every second input signal to a. The primitive returns to its original state upon outputting this signal on b. If the primitive is in its original state and receives only one input signal from path a, no output ensues. The primitive can be reset to its original state (other than by receiving a second signal to a) by a reset signal to input path r, which will produce an output signal on path k. The R-Counter is in fact a so-called Resettable Join module [60] of which the two regular input paths have been fused into a single input path a. Unlike the conventional resettable join, the R-Counter is capable of *arbitration* when conflicting input signals arise. Arbitration is the functionality according to which one out of a number of

TABLE II
TRANSITION RULES REQUIRED FOR SIMULATING DELAY-INSENSITIVE CIRCUITS ON THE CELLULAR ARRAY. A CONFIGURATION IN THE CELLULAR ARRAY MATCHING THE LEFT-HAND SIDE OF A RULE IS TRANSFORMED INTO THE CONFIGURATION ON THE RIGHT-HAND SIDE

1) 	**Rules for Signal propagation (1) and the Fork (2)**	2) 
3) 	**Rules for the Merge (3.4)**	4) 
5) 	**Rules for the R-Counter (5.6)**	6) 

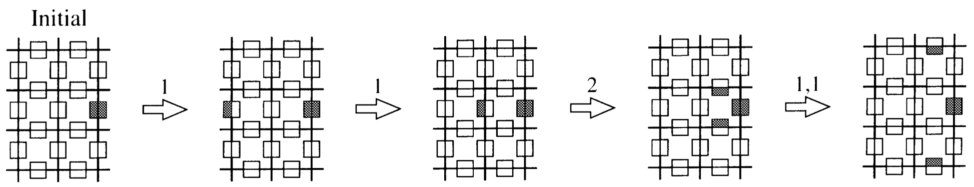


Fig. 4. Sequence of configurations in which a Fork receives one signal on its input path, after which it produces one signal to each of its two output paths. The initial configuration without input signals is labeled as such. For each time a transition rule is used, its label in Table II appears above the corresponding arrow.

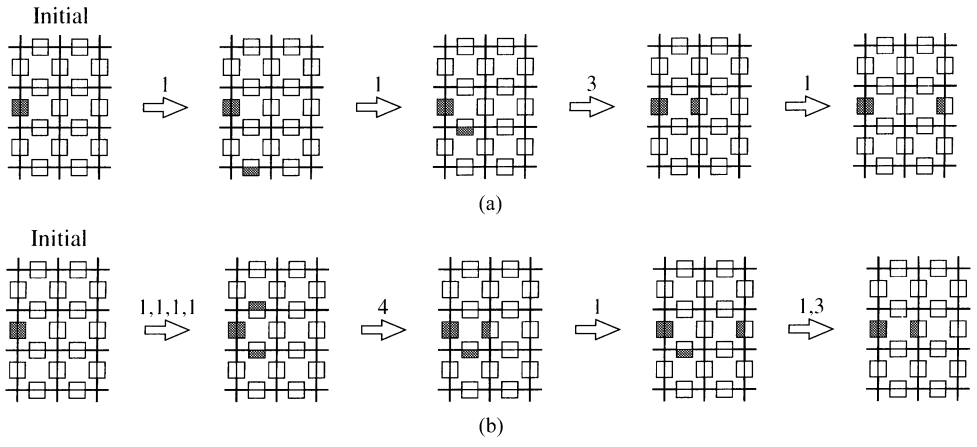


Fig. 5. Sequence of configurations in which a Merge receives (a) one signal at one of its input paths and (b) two simultaneous signals at its input paths. All signals are redirected toward the output path.

competing parallel processes is assigned the exclusive access to a shared resource. This functionality is useful for building a *sequencer* [47], a device that can arbitrate between two input signals (see also [13], [55]). The R-Counter is represented by a configuration of three cells of which only two cells may undergo transitions and the other cell is merely used to hold the configuration together. If an input signal arrives from path a, the configuration enters a special state due to rule 5, like in the third configuration in Fig. 6(a). If one more input signal arrives from path a, the configuration returns to its original state and an output signal is produced on path b due to rule 6, like in the second half of Fig. 6(a). If an input signal arrives from path r, like in Fig. 6(b), it stays pending until a signal arrives from path a. The signal on a again puts the configuration in the special state due to rule 5, after which the signal from r enters the configuration, an output signal is produced on path k, and the configuration is restored to its initial state, due to rule 2. If the order by which signals a and r arrive is the other way around, basically the same happens. If there are two input signals to a and one to r (not shown in figures), one of the cases in Fig. 6 happens, and one signal remains pending, on either a or r, depending on the case.

Since most circuits laid out on a 2-D cellular array have paths that cross each other, we need a configuration for crossing

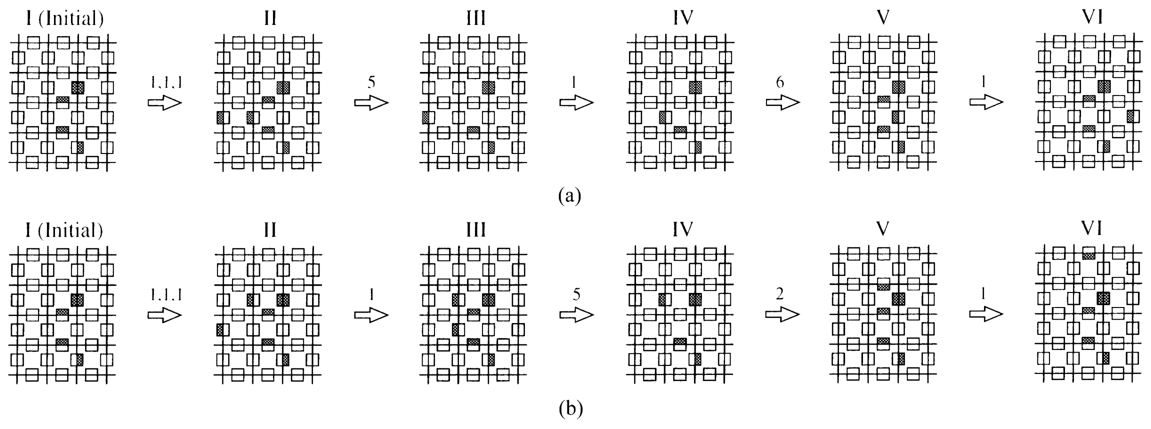


Fig. 6. Configurations of an R-Counter receiving various sequences of input signals. (a) Two subsequent signals are received from input path a. Upon receiving the first signal, the module goes into a special state (configuration III), but outputs no signals. Upon receiving the second signal, the module produces one signal on its output path b, in the process returning to its original state. (b) If the module receives one signal from each of its input paths r and a, it produces one signal on its output path k. Even if the signal from r arrives first at the configuration, it is kept pending until the signal from a arrives.

signals. The cells at a crossing of two paths are a shared resource, so arbitration is required to assign this resource in a certain order to signals on the paths. Following the same approach as in [13], [16]–[18], we construct a circuit for this task that is called a *crossing sequencer* [see Fig. 7(a)]. In this design a signal continuously runs around (busy waiting) to check for the presence of signals that want to cross each other. This circuit lacks *fair arbitration*, in the sense that it is unable to guarantee that signals arriving at both input paths at around the same time will be allowed to pass through with the same probability. Rather, if there are many input signals piling up at both input paths, the signals from one input path tend to be processed first, followed by the signals from the other input path—a shortcoming that is not problematic as in practice crossings with such busy traffic are rare. A more efficient but complicated version of this circuit, in which no signal is required to run around in the absence of input signals, and in which arbitration is fair, can be derived from the design of the sequencer module in [47]. The circuit implementation of the crossing sequencer on the asynchronous cellular array is shown in Fig. 7(b). Based on this design, any arbitrary delay-insensitive circuit can be laid out on our asynchronous cellular array. In practice, most crossings in a circuit implementation can be accomplished without the use of the crossing sequencer due to information available about the order in which signals are to cross. If no such information can be deduced from the circuit design, however, the crossing sequencer is the only option to make signals cross.

In Fig. 8, we show a circuit design (based on [47]) and its implementation on the asynchronous cellular array of a delay-insensitive module called a *TRIA* [58]. It has three input lines and three output lines, each output line corresponding to a unique pair of input lines. If a TRIA receives one input signal on one input line, it stays pending until a second input signal arrives on another input line. It then responds by outputting a signal to the corresponding output line. The TRIA module is very useful for the efficient design of delay-insensitive circuits, witness for example its use in the designs in Fig. 9.

The NAND gate, commonly used in synchronous circuits, has a delay-insensitive counterpart that conducts the logical NAND operation even if input signals are delayed. To represent a binary signal in a delay-insensitive circuit, two lines are usually employed, one labeled 0 and one labeled 1. Called *dual-rail encoding*, this method encodes a 0 by a signal on the line labeled 0, and a 1 by a signal on the line labeled 1. Signals on both lines at the same time are forbidden, and the absence of a signal on either line indicates that no information is being transmitted. Using dual-rail encoding for its inputs and outputs, the NAND-gate requires four input lines, i.e., a_0 and a_1 encoding one input signal and b_0 and b_1 encoding the other input signal, as well as two output lines, encoded by c_0 and c_1. The design of the delay-insensitive NAND-gate, based on [55], is given in Fig. 9(a). Though we do not give the layout of this circuit on the cellular array to save space, the configuration can be easily designed from the configurations of the Merge, the Fork and the TRIA. The delay-insensitive circuit can be laid out on the asynchronous cellular array as such without using the crossing sequencer in Fig. 7, since all line crossings are designed to be collision-free.

Fig. 9(b) shows the design of a delay-insensitive 1-bit memory, which, unlike the design in [13], is expressed in terms of the Merge, the Fork, and the TRIA. Again dual-rail encoding is employed. To write the value 0 or 1 into the memory, a signal is input to the line W_0 or W_1, respectively, after which an acknowledgment signal emerges from line A_0 or A_1. To read the contents of the memory, a signal is input to line R, after which a signal emerges from line R_0 or R_1, depending on the contents of the memory. A crossing sequencer is not required for implementing this design on the cellular array.

Higher order structures, like counters and for-loops, can be constructed from a delay-insensitive 1-bit memory in the way lined out in [13]. A delay-insensitive NAND-gate may be used for evaluating Boolean expressions. It is not recommended to apply the delay-insensitive NAND-gate in the same way as a conventional NAND-gate to build logic circuits, because it tends to require an increased amount of hardware resources.

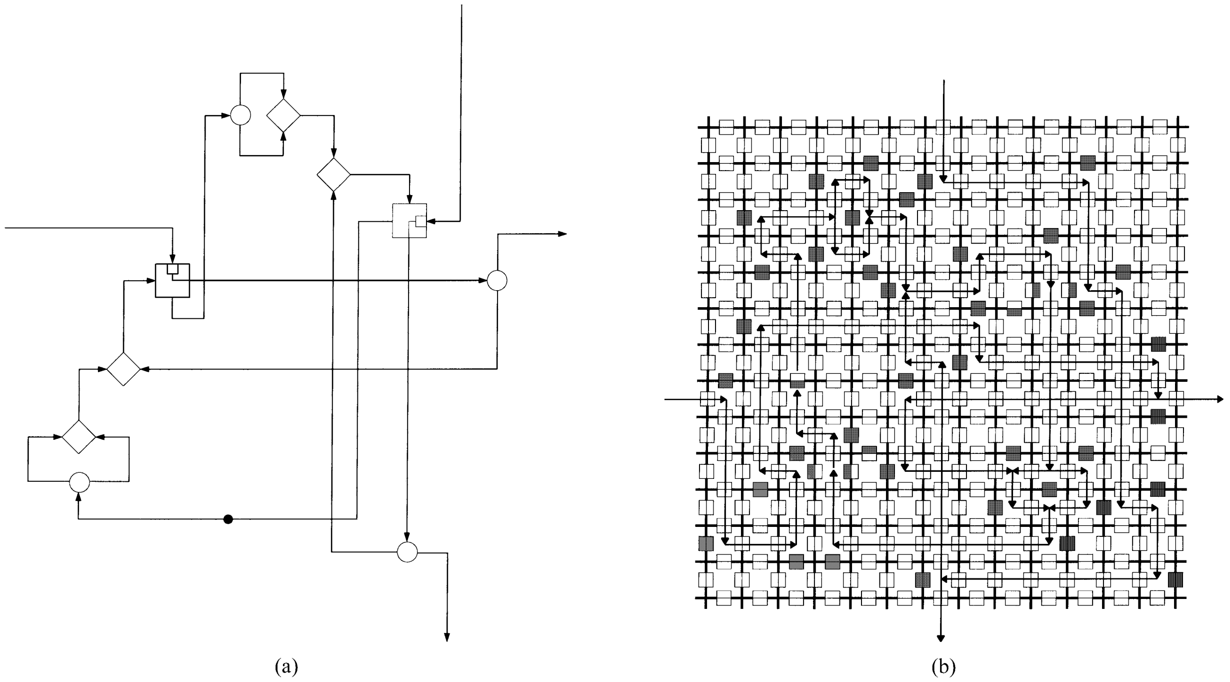


Fig. 7. (a) Delay-insensitive circuit scheme for crossing signals without collisions, and (b) its implementation on the asynchronous cellular array. The arrows in the cellular array denote paths via which signals travel between primitives, or via which signals are input or output. The circuit contains two R-Counters 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 the R-Counters alternatingly and produces an output signal at the corresponding path if the R-Counter is in a state denoting the arrival of an input signal.

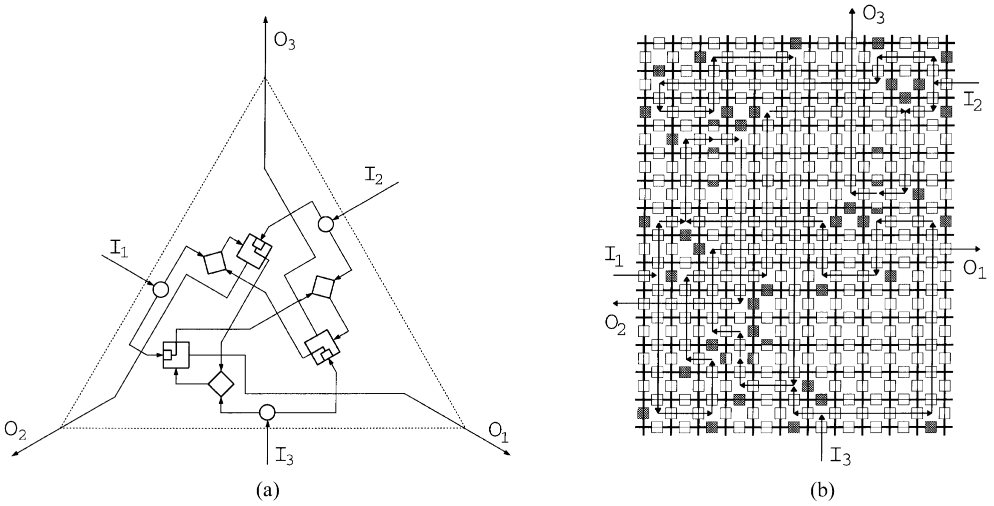


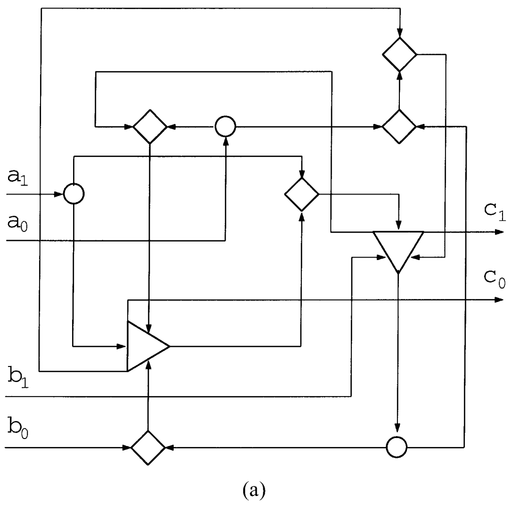
Fig. 8. (a) Delay-insensitive circuit scheme for a TRIA module based on the design in [47], and (b) its implementation on the asynchronous cellular array. When a TRIA receives an input signal from each of the two paths $I_i (i \in \{1, 2, 3\})$ and $I_j (j \in \{1, 2, 3\} \setminus \{i\})$, it outputs a signal to the line O_{6-i-j}. The circuit is laid out on the asynchronous cellular array such that the path crossings are collision-free, so the crossing-sequencer in Fig. 7 is not required.

For example, though a delay-insensitive 1-bit memory can be constructed from delay-insensitive NAND-gates, Fork, and Merge modules, the design will be much more complicated than a direct design in terms of TRIA, Fork, and Merge modules. Similarly, it is much more efficient to build combinational circuits like counters, adders, multipliers, etc., directly from the delay-insensitive primitives in Table I than to use the delay-insensitive NAND-gate to build them.

V. Fault-Tolerance

The cellular array proposed in the previous section is not fault-tolerant: if a cell misoperates, for example through a mistake in a transition or some bits of a memory flipping due to noise, no recovery is possible, and the cellular array ceases to operate in the way intended. In this section we equip the cellular array with fault-tolerance, using techniques from error correcting coding theory (e.g., see [37]).

Adopting a common convention, we denote an error correcting code by the 3-tuple (n, m, d), whereby n is the word length, expressed in bits, m is the number of codewords, and d is the minimal distance between any two codewords, i.e., the minimum number of bits by which any two codewords differ. We recall that a code with the minimum distance d can *correct* up to $\lfloor (d-1)/2 \rfloor$ errors. If d is even, it can in addition *detect*



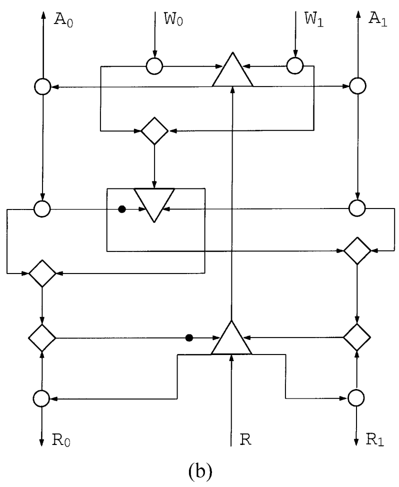


Fig. 9. (a) Circuit scheme for a dual-rail encoded delay-insensitive NAND-gate. The lines a_0 and a_1 correspond to one input line in a conventional NAND-gate, the lines b_0 and b_1 to the other input line, and the lines c_0 and c_1 correspond to the output line. If a signal is input to each of the two lines in one of the sets $\{a_0, b_0\}, \{a_0, b_1\}$, or $\{a_1, b_0\}$, a signal is output to c_1. If a signal is input to each of the lines a_1 and b_1, a signal is output to c_0. If there is only one input signal, it becomes pending, and the gate will wait for the second input signal to arrive, before producing an output signal. Other combinations of input signals are illegal. (b) Delay-insensitive circuit scheme for a dual-rail encoded 1-bit memory. 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 lines to two TRIA modules. A signal pending on the left line denotes 0, the initial state, and a signal on the right line denotes the state 1. Writing a 0-bit or 1-bit into the memory is done by sending a signal to one of the input lines W_0 and W_1 respectively. This deletes the two pending signals storing the memory state, after which the new contents is written into the memory, and an output signal for acknowledgment is produced on either A_0 or A_1, respectively. By sending a signal to line R, the memory is read out, and a signal is produced on either line R_0 or R_1, depending on the state of the memory.

$d/2$ errors. To design an efficient error correcting code given a certain value of n, the value of both d and m should be as large as possible, but these are conflicting requirements. This paper concerns only *linear codes*, which have the property that the bit-wise addition—modulo 2 in case of binary codes—of any two codewords produces again a codeword.

In a previous paper [46] we make an asynchronous cellular array fault-tolerant by taking all the bits of all the memories associated with a cell, and encoding them as codewords in an error correcting code based on a ternary number system. Using this construction, in each memory up to one error can be corrected (and two errors detected), and (in principle) in each cell an invalid combination of states of the memories associated with it can be detected. A disadvantage of this approach is that it is difficult to find codes for which the codewords encode the cell configurations representing primitives and signals. In [46] this problem is resolved by adjusting the configurations such as to match a suitable code. The cellular array in [46], however, is simpler than in the current paper: it is so simple that it allows only one signal at a time to run around in a circuit laid out on it. The richer set of configurations of the cellular array in the current paper makes this approach harder to pursue. Another disadvantage of the approach in [46] is that the correction of errors is complicated, requiring complex circuitry in the cells. This problem is resolved in [46] by correcting the errors of each memory individually, rather than taking all memories associated with a cell together in the error correction. To this end, only the subcodes associated with individual memories need to be considered, which is a substantial simplification, though it goes at the cost of loosing the ability to correct or detect invalid combinations of states of memories associated with a cell. In the current paper we extend this approach, in a way that not only single errors can be corrected in the memories but also multiple errors. We start with memories that can correct up to one bit error, and then address the more general case of multiple errors. We also discuss a variation based on a repetition code and compare it with the proposed codes.

The messages to be encoded by an error correcting code are all possible states of one memory of a cell. As each memory contains two bits, there are four possible memory states: 00, 01, 10, and 11. This translates into four codewords, so $m = 4$. To encode these messages, it is necessary to decide on the error correcting ability of each memory, i.e., the maximum number of corrupted bits in a memory that can be recovered. A code that is able to correct one error needs to have a minimal distance of $d = 3$. Given a number of words of $m = 4$, a lower bound for the word length n of a binary code can be deduced from the *Plotkin bound* (e.g., see [37]), which states that if $n < 2d$ then the following equations should hold for binary codes:

$$m \leq 2\left\lfloor \frac{d}{2d-n} \right\rfloor \quad \text{if } d \text{ is even,} \tag{1}$$

$$m \leq 2\left\lfloor \frac{d+1}{2d+1-n} \right\rfloor \quad \text{if } d \text{ is odd.} \tag{2}$$

This bound is violated for the parameter values $m = 4, d = 3$, and $n = 4$, but not when n is increased to $n = 5$, so at least five bits are required as the word length to allow the correction of single bit errors. Indeed, it is possible to construct a code with these parameters, namely the code consisting of the codewords 00000, 11100, 00111, and 11 011.[3] In terms of memories, these codewords are displayed like in Fig. 10. It is easily verified that flipping any arbitrary single bit in any of these codewords results

[3]This code can be constructed from a so-called *Hamming code* (e.g., [37]) with parameters $n = 7, m = 16$, and $d = 3$, by selecting the four codewords of which the first two bits are zero and removing these two bits.

in a word that has distance 1 to the original codeword, and distance at least 2 to each of the other codewords. In other words, even if one bit is corrupted, it can be uniquely determined what the original codeword should have been.

The above code can be generalized to codes with different lengths that correct more than one error. To this end, we observe that codes with a high minimum distance tend to have bits of which the values 0 and 1 are evenly distributed among the codewords for each bit. In our case this translates into each bit being 0 for two codewords and 1 for the other two codewords. Assuming that one of the codewords is the all-zero codeword, we then obtain the scheme in Fig. 11 for a code with four words. Let the number of bits in groups a, b, and c in Fig. 11 be n_a, n_b, and n_c, respectively, then $n = n_a + n_b + n_c$. From this scheme it is deduced that the distance between for example the first and the second codeword is $n_a + n_b$. The minimum distance d of the code is the minimum of the distances of all pairs of codewords, so, $d = \min(n_a + n_b, n_b + n_c, n_a + n_c) = \min(n - n_c, n - n_a, n - n_b)$. To maximize d the differences between n_a, n_b, and n_c should thus be as small as possible. We then consider three cases, depending on the remainder obtained when n is divided by 3 as follows:

- $n = 3p$ for some p. The distance will be maximal if $n_a = n_b = n_c = p$. In this case $d = 2p$, which implies that up to $p - 1$ errors can be corrected and p errors detected in a word.
- $n = 3p + 1$ for some p. The distance will be maximal if two groups have length p and one group has length $p + 1$, so let $n_a = n_c = p$ and $b = p + 1$. In this case $d = n - (p + 1) = 2p$, which implies that up to $p - 1$ errors can be corrected and p errors detected in a word.
- $n = 3p + 2$ for some p. The distance will be maximal if two groups have length $p + 1$ and one group has length p, so let $n_a = n_c = p + 1$ and $n_b = p$. In this case $d = n - (p + 1) = 2p + 1$, which implies that up to p errors can be corrected in a word.

By substituting the values of n and d in terms of p into (1) or (2), it is easily deduced that $m \leq 4$, and furthermore, that d is the largest value for each given n for which this holds, implying that the above codes are optimal with respect to the Plotkin bound. Asymptotically, up to almost one third of a memory's bits can be corrected using these codes.

As an example, we deduce that a code with $n = 14$, $d = 9$, and $m = 4$ has codewords 00000000000000, 11111111100000, 00000111111111, and 11111000011111. Rearranging the bits of the codewords gives the scheme in Fig. 12. This code can correct up to four arbitrary errors in a memory of 14 bits, but even a better performance is possible, as will become clear later in this section.

We compare this code with a so-called (14, 4, 7) *repetition code*, in which words are represented by seven pairs of bits, each pair being a copy of the original message (see Fig. 13). Correction of errors in the bits is done by counting the occurrence of each of the four messages 00, 01, 10, and 11 among the bit pairs, and setting all bits according to the bit pair with the highest count, a method called *majority voting*. Obviously, three errors can always be corrected, wherever they occur, because the

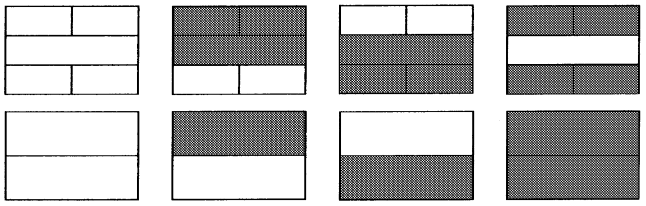


Fig. 10. Graphical notation of a (5,4,3)-code. The top row contains the codewords, the bottom row the original messages. A dark block encodes a bit with value 1, a white block a bit with value 0. This code allows the correction of one arbitrary bit error.

$$\begin{array}{ccc} \overbrace{0...0}^{a} & \overbrace{0...0}^{b} & \overbrace{0...0}^{c} \\ 1...1 & 1...1 & 0...0 \\ 0...0 & 1...1 & 1...1 \\ 1...1 & 0...0 & 1...1 \end{array}$$

Fig. 11 Scheme of a code with four codewords, one of which contains all 0 bits. Each bit position (column) contains two 0-bits and two 1-bits. The bits are arranged in three different groups a, b, and c, whereby the pattern of bits within a group as seen over the four words is identical.

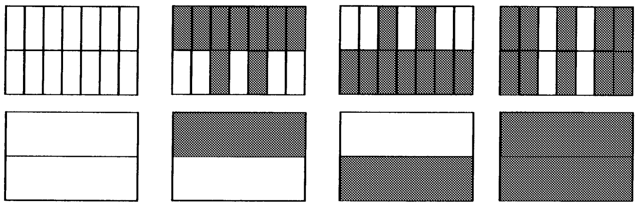


Fig. 12. Graphical notation of a (14, 4, 9)-code. The top row contains the codewords, the bottom row the original messages. A dark block encodes a bit with value 1, a white block a bit with value 0. The bits of the codewords have been shuffled to make for an easy-to-discern graphical representation. This code allows the correction of up to four arbitrary bit errors.

remaining four bit pairs—the majority—remains uncorrupted. More in general, for a memory with $n = 2r$ bit pairs for some odd $r, (r - 1)/2$ arbitrary errors can be corrected. We can do better, however, for some combinations of errors. For example, if in the case $r = 7$ all bits in three bit pairs are corrupted, making for a total of six bit errors, still the bit pairs carrying the original message are in the majority, allowing the correction of all errors. A straightforward analysis of the cases with four, five, and six errors that can be resolved by the majority voting strategy reveals that about 65% of the cases with four erroneous bits can be resolved, about 52% of the cases with five erroneous bits, and about 29% of the cases with six erroneous bits.

How are errors corrected under the scheme of Fig. 11? An important requirement is that the correction algorithm should be straightforward to keep the cells and memories simple. The most common way to correct errors for codes that are linear—which all our codes are—is by computing its syndrome (e.g., see [37]) and deducing from it the bit positions of the errors. While this is relatively straightforward for single-error correction by codes like the (5,4,3)-code in Fig. 10, it is more complicated for codes with larger distances, and it may result in cells and memories becoming too complex. For our (14,4,9)-code we show a different method for error correction, which employs a table in which the four codewords are stored. Given a word representing

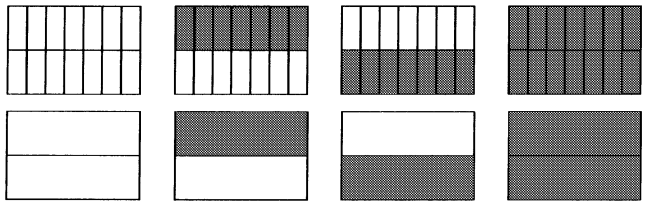


Fig. 13. Graphical notation of a (14, 4, 7) repetition code. The top row denotes the codewords, the bottom row the original messages. A dark block encodes a bit with value 1, a white block a bit with value 0. This code allows the correction of up to three arbitrary bit errors.

a memory that may have to be corrected, the method calculates the distances to the table entries, and selects the codeword with the smallest distance as the most likely. Called *distance comparison* in this paper, this method has limited complexity due to the small number of codewords, and it comes with the bonus that in many cases five and in some cases even six errors can be corrected. For example, if there are errors in the first three bits, the sixth bit, and the tenth bit of the all-zero word 00000000000000, it becomes 11100100010000, and this word has distance 5 to the all-zero codeword, but distances 6, 10, and 7 to the codewords 11111111100000, 00000111111111, and 11111000011111, respectively.

Some six bit errors are also recoverable. For example, if bits 1, 2, 6, 7, 10, and 11 are erroneous in the all-zero word, we obtain 11000110011000. This word has distance 6 to the all-zero codeword, and distances 7, 7, and 8 to the codewords 11111111100000, 00000111111111, and 11111000011111, respectively.

Along the same lines, all the cases of five and six errors can be listed up systematically, revealing that about 75% of the cases with five erroneous bits and about 20% of the cases with six erroneous bits can be resolved when using distance comparison (for details see [61]).

VI. Implementation Issues

As the memories of a cellular array design increase in size with their tolerance to faults, the cells grow in complexity. To find suitable implementations of fault-tolerant cellular arrays, it is important that the presumptions are adhered to that prompted us to use cellular arrays in the first place. That is, cells should be as simple as possible, and the underlying structure should be as regular as possible on a fine-grained level. Additionally, the occurrence of errors in the cellular array should be localized in the sense that errors only affect a small part of the outcome, even over a longer time frame. These requirements suggest the further subdivision of cells in independent units that are similar to and independent of each other. In this section, we investigate how such an organization can be obtained, and in particular investigate how a cell's mechanism for carrying out transitions can be implemented as simple identical units that are independent of each other.

The actions taking place in a cellular array consist of the following stages.

1) Testing whether the states of the memories associated with a cell can be matched to the left hand side of a transition rule.
2) If a match is possible, the corresponding memories are locked such that they can be exclusively used by the cell for carrying out the transition. In case a transition rule is divided into separate units, each unit carries out its transition independently of the other units. Units may carry out their transitions at different times, but they should all start after a match is made. Furthermore, only after all units of a cell finish, will the transition be considered finished, and the memories be unlocked.
3) Errors in each memory are corrected at random times, provided the memory is not locked by a cell carrying out a transition.

Under this scheme, bit errors in the memories associated with a cell will disable all transitions of the cell, because a match between the memory states and the left hand sides of the transition rules cannot be made in the presence of errors. As soon as all bits of the memories are corrected, however, the cell will automatically continue its transitions.

How can we distribute the mechanism to carry out transitions over independent units? Starting with the (14,4,7) repetition code, we divide a memory up into seven pairs of bits, just as we did in the previous section. The mechanism to carry out transitions on a cell can then be divided into units, each acting on independent groups of bit pairs, like in Fig. 14. A transition acting on a group of bit pairs is called a *group-wise transition*. The rules governing group-wise transitions are the same as those in Table II, because the state of each group of bit pairs is identical to the state of the memory in the original nonfault-tolerant cellular array. Though this method requires seven units to carry out transitions, the units are all identical, which facilitates regularity, and more importantly, the units are independent: for example if one of the transition units errs, only one of the bit groups in each memory associated with a cell will be affected, and this can be recovered by the error correction mechanism in each memory.

A similar method applies when other error correcting codes are used, such as the (5,4,3)-code and the (14,4,9)-code, though it is more difficult to find independent units to carry out transitions. When dividing a memory into independent groups of bits, it is important that there is a one-to-one correspondence between the states of each group on the one hand, and the memory states of the original nonfault-tolerant cellular array on the other hand, to ensure that it can be uniquely determined which transition rule applies for each combination of group states. So, in the above example of the (14,4,7) repetition code it would be impossible to design transition rules acting on only one bit of each of the four memories, because it would result in rules of which some have identical left hand sides—a situation not allowed in our model. A one-to-one correspondence for the (5,4,3)-code and the (14,4,9)-code can be obtained by grouping the memory bits like in Fig. 15(a) and (b), respectively. We investigate this in more detail for the (14,4,9)-code. The case of the (5,4,3)-code, though similar, is not very practical, because only two groups of bits can be distinguished in each memory in this case and these groups have different numbers of bits. With respect to the (14,4,9)-code, consider the four states of a memory at a cell's upper edge in Fig. 16 (original). These memory states are encoded by the

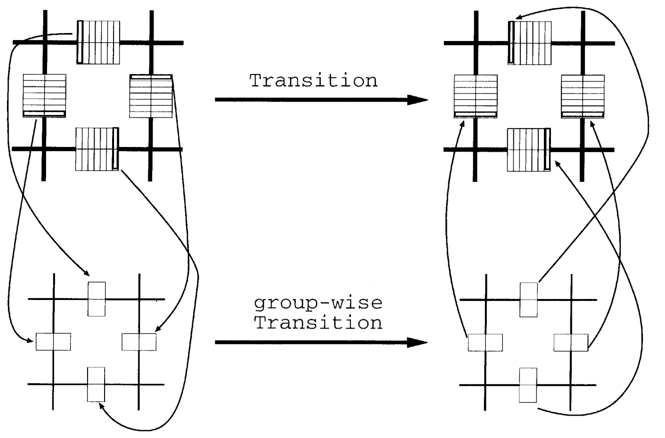


Fig. 14. The transition mechanism in a fault-tolerant cellular array can be distributed over independent units that act in similar ways. The upper part of the figure shows a transition as it would take place on the memories associated with a cell, and the lower part shows its implementation for each quartet of bit pairs independently.

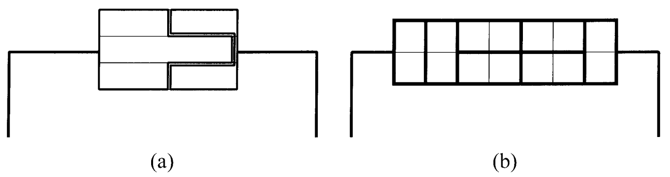


Fig. 15. (a) Memory bits of a (5,4,3)-code divided into two independent groups, one with three bits and one with two bits. (b) Memory bits of a (14,4,9)-code divided into seven independent groups, all with two bits. Some groupings are horizontal and some vertical to ensure a one-to-one correspondence with the memory states in the original nonfault-tolerant cellular array (see Fig. 16).

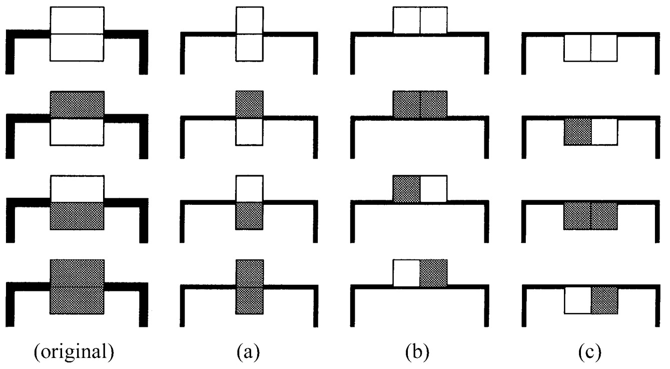


Fig. 16. (Original) The states of a memory at a cell's upper edge in a nonfault-tolerant cellular array. (a) The corresponding states of a vertical pair of bits in the fault-tolerant cellular array based on the (14,4,9)-code. (b) The corresponding states of a horizontal pair of bits on the outer edge of a cell. (c) The corresponding states of a horizontal pair of bits on the inner edge.

codewords of the (14,4,9)-code, as in Fig. 12. The grouping of the 14 bits in the codewords according to Fig. 15(b) then results in three different patterns, namely the bit pairs in Figs. 16(a), (b), and (c). There is a one-to-one correspondence between the patterns in each of (a), (b), and (c) on the one hand, and the original memory states in Fig. 16 (original) on the other hand, implying that transition rules can be formulated for the corresponding groups.

Fig. 17 shows the three group-wise transition rules corresponding to the transition rule for signal propagation (rule 1 in Table II rotated counterclockwise by 90°). To propagate a signal over a fault-tolerant cellular space based on the (14,4,9)-code, all three rules acting on the groups of bit pairs in Fig. 16 are applied in ensemble. The rule under (a) resembles the original transition rule, but the rules under (b) and (c) do not. However, the mechanisms implementing them are very similar. The above discussion suggests that it is possible to implement the mechanism for transitions in a distributed way, such that independent units are very similar and independent with respect to the occurrence of errors. The similarity notwithstanding, the three different types of bit group in Fig. 16(a), (b), and (c), respectively, require different transition rules, which translates into less regularity than with the use of the (14,4,7) repetition code, and, depending on the implementation, it may also require more memory in a cell.

VII. Conclusion and Discussion

Asynchronous cellular arrays have many advantages to offer as architectures for computers realized by nanotechnology. Their regular structure paired with the absence of a central clock may facilitate efficient physical implementations and manufacturing methods. Making cells as simple as possible, a key goal, is an ongoing process of experimenting with new designs of asynchronous circuits and cellular arrays. The cellular array constructed in this paper is based on a novel type of delay-insensitive circuit [47]. This allows cells to be less complex than in previously proposed models, as measured in terms of the number of bits to represent the table of transition rules in a cell as well as a cell's state (see [13]). To store one transition rule we need 16 bits, since both the left hand side and the right hand side of the rule represent four memory states of two bits. For six transition rules a total of $6 \times 16 = 96$ bits is thus required. To store the states of a cell we need four memories, but the two bits in each memory are shared between the two cells associated with the memory, so an average of four bits per cell is required. We thus have a total of $96 + 4 = 100$ bits per cell. This compares to 148 bits per cell in the asynchronous cellular array in [13], which employs nine transition rules. We believe a further reduction in the cell complexity may be possible, perhaps down to a level of only four transition rules, which would translate to 68 bits per cell. A four-rule asynchronous cellular array has been proposed in [53], but, though it can conduct the same class of computations as conventional computers, it does so with an extremely low efficiency, as only one signal at a time may move around in all of the circuitry simulated on the cellular array. So, the hunt is on for squeezing more functionality out of a limited number of transition rules.

Physical implementations of nanocomputers faces many difficulties. There are defects and faults arising from the instability and noise-proneness on nanometer scales, which may lead to unreliable and undesirable results of computation. In order to ensure more reliable computation, techniques are necessary to cope with such errors. The cellular array constructed in the first part of this paper is made fault-tolerant by representing the memory states as codewords in an error correcting code. Provided efficient codes are used, these techniques requires less

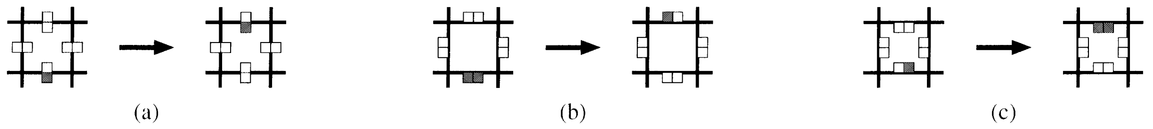


Fig. 17. Group-wise transition rules corresponding to the transition rule for signal propagation (rule 1 in Table II, rotated counterclockwise by 90 degrees). The rules in (a), (b), and (c) carry out transitions on the groups of bits in Fig. 16(a), (b), and (c), respectively.

hardware redundancy than previously proposed schemes, like R-fold modular redundancy (see Introduction), which amount to repetition codes, of which errors are corrected by majority voting. Our scheme allows correction even if up to asymptotically one third of the bits is corrupted. In contrast, a repetition code allows only up to asymptotically one fourth of the bits to be corrupted, witness for example the worst case in which just more than half of the repeated bit pairs in a codeword are corrupted by one bit error each, all of them in the same position of the bit pair: in this case just over one fourth of the bits is corrupted, but no recovery is possible.

If the errors are in certain favorable positions, more bit errors can be corrected in both schemes (see Section V). At best, recovery is then possible if almost half of the bits are corrupted. Our scheme tends to perform better in most of such instances than the repetition code scheme. The ability of both coding schemes for correcting errors beyond their distances indicates that the codewords, when represented as points in hyperspace, have much space around them in selected bit positions. This phenomenon is mainly due to the limited number (four) of codewords. A comparison can be made with packing a three dimensional space with balls. It is harder to do this efficiently with a few big balls than with a large number of smaller balls. An upper bound for the number of balls with a certain diameter that can be packed in an n-dimensional hyperspace is the so-called *Hamming bound* (e.g., see [37]). This bound is not tight for our proposed coding scheme, and the situation is even worse for repetition codes. Even so, no improvement can be expected with our codes, because another bound, the Plotkin bound, is tight for them, as pointed out in Section V.

Both the schemes for the (14,4,9)-code and the (14,4,7)-code in Section V are unable to correct more than six errors, but they can be used to detect them, as can be shown along the same lines. Error detection, however, is hardly useful in our context, for once a memory's errors become irrecoverable, it is difficult to restore it by requesting resubmission—a strategy that works in communications.

Though we have constructed error correcting codes with word lengths 5 and 14, any word length above 4 may be chosen. The choice for a particular word length is determined, apart from how big a memory may be, by the probability of errors occurring and their type. For example, errors occurring in bursts require longer codewords.

Inevitably, redundancy is required to attain fault-tolerance, but this results in more complex cells. More complexity tends to translate into less regularity, unless a way can be found to divide a cell into identical parts. We have proposed a method to distribute a cell's mechanism for carrying out transitions into units that have about the same complexity as a cell's transition mechanism in a nonfault-tolerant cellular array (see Section VI). Each unit acts on two (or three) bits of each of the cell's four memories in an operation that we call a group-wise transition. As the units are independent of each other, an error occurring in one of them does not affect the others, so errors occurring in group-wise transitions remain localized and can be corrected by an error correcting mechanism acting on the memories, provided that the number of errors lies within the correction ability of the code. A unit may be implemented as a separate circuit, in which case all units can work in parallel. A unit may also be shared among all groups in a cell, in which case one unit carries out all group-wise transitions of a cell one by one. This sequential way of processing has as disadvantage that correlations between errors during different group-wise transitions may arise and that execution of transitions is slower. The advantage is that less hardware is necessary, though an additional mechanism may be required to coordinate the sequential execution of the group-wise transitions.

In this paper, we have implicitly assumed that the circuits used for error correction are error free. When implementing cells on the nanometer scale this assumption is easily violated. The best way to cope with this problem is to design the circuits such that any error in the error-correcting circuit has only impact on a limited number of bits, preferably one bit at a time. Erroneous bit states caused by transient errors in the error-correcting circuit may then be corrected in the next iteration through the circuit.

Though this paper shows how to efficiently simulate delay-insensitive circuits on asynchronous cellular arrays in a fault-tolerant way, it leaves open the challenge of configuring the inherently homogeneous cellular hardware into particular delay-insensitive circuits [12]. 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. A method suggested in [13] for this uses the collective behavior of the cells to copy information from one part of the cellular array to another part, in a way resembling self-reproduction in cellular arrays [48], [49], [62]. To implement configurability on an asynchronous cellular array, memories may require more bits to carry configuration information. This implies that a fault-tolerant version of the cellular array will require error correcting codes with more codewords than the four assumed in this paper. These codes are likely to have better efficiency than the codes in this paper, because, being based on more balls corresponding to the codewords, the hyperspace will be filled more efficiently. The Hamming bound will be approached more closely, and related to that, the number of bits required to make the memories fault-tolerant may hardly have to increase, as compared to the proposed codes.

Defect-tolerance, though important for nanotechnology applications, has not been discussed in this paper. Some results on defect-tolerance have been obtained by building the *Teramac* [27], a parallel computer based on field-programmable gate arrays (FPGAs) that is able to achieve high-performance computing, even if a significant number of its components are defective. As this method requires a master computer setting up a table of routes around defects and configuring the hardware correspondingly, it cannot be performed autonomously by the defective computer, and as such it may be less suitable for certain types of nanocomputers. In the context of cellular arrays, some results on defect-tolerance are given in [63]–[65], but they focus mainly on variations of the *Firing Squad Synchronization Problem*, a synchronization problem on synchronous cellular arrays that is mainly of theoretical interest.

VIII. Acknowledgments

The authors would like to thank K. Morita of Hiroshima University, Japan, for his support and discussions.

References

[1] A. Bachtold, P. Hadley, T. Nakanishi, and C. Dekker, "Logic circuits with carbon nanotube transistors," *Science*, vol. 294, pp. 1317–1320, 2001.

[2] C. P. Collier, E. W. Wong, M. Belohradsky, F. M. Raymo, J. F. Stoddard, P. J. Kuekes, R. S. Williams, and J. R. Heath, "Electronically configurable molecular-based logic gates," *Science*, vol. 285, pp. 391–394, 1999.

[3] Y. Cui and C. Lieber, "Functional nanoscale electronic devices assembled using silicon nanowire building blocks," *Science*, vol. 291, pp. 851–853, 2001.

[4] Y. Huang, X. Duan, Y. Cui, L. J. Lauhon, K. Kim, and C. M. Lieber, "Logic gates and computation from assembled nanowire building blocks," *Science*, vol. 294, pp. 1313–1317, 2001.

[5] H. W. Ch. Postma, T. Teepen, Z. Yao, M. Grifoni, and C. Dekker, "Carbon nanotube single-electron transistors at room temperature," *Science*, vol. 293, pp. 76–79, 2001.

[6] M. A. Reed, J. Chen, A. M. Rawlett, D. W. Price, and J. M. Tour, "Molecular random access memory cell," *Appl. Phys. Lett.*, vol. 78, pp. 3735–3737, 2001.

[7] T. Rueckes, K. Kim, E. Joselevich, G. Y. Tseng, C. Cheung, and C. M. Lieber, "Carbon nanotube-based nonvolatile random access memory for molecular computing," *Science*, vol. 289, pp. 94–97, 2000.

[8] W. Porod, "Nanoelectronic Circuit Architectures." *Handbook of Nanoscience, Engineering, and Technology*, W. A. Goddard III, D. W. Brenner, S. E. Lyshevski, and G. J. Lafrate, Eds. Boca Raton, FL: Chemical Rubber Company Press, 2002, ch. 5.

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

[10] M. Biafore, "Cellular automata for nanometer-scale computation," *Physica D*, vol. 70, pp. 415–433, 1995.

[11] J. Lyke, G. Donohoe, and S. Karna, "Reconfigurable Cellular Array Architectures for Molecular Electronics," Air Force Research Laboratory, Tech. Rep. AFRL-VS-TR-2001-1039, 2001.

[12] L. J. K. Durbeck and N. J. Macias, "The cell matrix: An architecture for nanocomputing," *Nanotechnology*, vol. 12, pp. 217–230, 2001.

[13] F. Peper, J. Lee, S. Adachi, and S. Mashiko, "Laying out circuits on asynchronous cellular arrays: A step toward feasible nanocomputers?," *Nanotechnology*, vol. 14, pp. 469–485, 2003.

[14] K. L. Wang, "Issues of nanoelectronics: A possible roadmap," *J. Nanosci. Nanotechnol.*, vol. 2, no. 3/4, pp. 235–266, 2002.

[15] K. Nakamura, "Asynchronous cellular automata and their computational ability," *Syst., Comput., Contr.*, vol. 5, no. 5, pp. 58–66, 1974.

[16] J. Lee, S. Adachi, F. Peper, and K. Morita, "Embedding universal delay-insensitive circuits in asynchronous cellular spaces," *Fundamenta Informaticae*, vol. 58, no. 3/4, pp. 295–320, 2003.

[17] S. Adachi, F. Peper, and J. Lee, "Computation by asynchronously updating cellular automata," *J. Stat. Phys.*, vol. 114, no. 1/2, pp. 261–289, 2004.

[18] L. Priese, "A note on asynchronous cellular automata," *J. Comput. Syst. Sci.*, vol. 17, pp. 237–252, 1978.

[19] S. Hauck, "Asynchronous design methodologies: An overview," *Proc. IEEE*, vol. 83, pp. 69–93, 1995.

[20] C. Joachim, J. K. Gimzewski, and A. Aviram, "Electronics using hybrid-molecular and mono-molecular devices," *Nature*, vol. 408, pp. 541–548, 2000.

[21] A. J. Heinrich, C. P. Lutz, J. A. Gupta, and D. M. Eigler, "Molecule cascades," *Science*, vol. 298, pp. 1381–1387, 2002.

[22] Y. Kameda, S. V. Polonsky, M. Maezawa, and T. Nanya, "Self-timed parallel adders based on DI RSFQ primitives," *IEEE Trans. Appl. Supercond.*, vol. 9, pp. 4040–4045, 1999.

[23] P. Patra, S. Polonsky, and D. S. Fussell, "Delay-insensitive logic for RSFQ superconductor technology," in *Proc. 3rd Int. Symp. on Adv. Res. in Asynchronous Circuits and Systems*. Las Alamitos, CA, 1997, pp. 42–53.

[24] C. S. Lent and P. D. Tougaw, "A device architecture for computing with quantum dots," *Proc. IEEE*, vol. 85, pp. 541–557, 1997.

[25] R. P. Cowburn and M. E. Welland, "Room temperature magnetic quantum cellular automata," *Science*, vol. 287, pp. 1466–1468, 2000.

[26] U. Golze and L. Priese, "Petri net implementations by a universal cell space," *Inform. Contr.*, vol. 53, pp. 121–138, 1982.

[27] J. R. Heath, P. J. Kuekes, G. S. Snider, and R. S. Williams, "A defect-tolerant computer architecture: Opportunities for nanotechnology," *Science*, vol. 280, pp. 1716–1721, 1996.

[28] J. von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms From Unreliable Components." *Automata Studies*, C. E. Sharmon and J. McCarthy, Eds. Princeton, NJ: Princeton Univ. Press, 1956, pp. 43–98.

[29] R. L. Dobrushin and S. I. Ortyukov, "Upper bound on the redundancy of self-correcting arrangements of unreliable functional elements," *Prob. Inform. Trans.*, vol. 13, pp. 203–218, 1977.

[30] N. Pippenger, "On networks of noisy gates," in *Proc. 26th IEEE Symp. on Foundations of Computer Science*, 1985, pp. 30–38.

[31] ——, "Invariance of complexity measures for networks with unreliable gates," *J. Assoc. Comput. Mach.*, vol. 36, no. 3, pp. 531–539, July 1989.

[32] P. G. Depledge, "Fault-tolerant computer system," *Inst. Elect. Eng. Proc.*, vol. 128, pp. 257–272, 1981.

[33] S. Spagocci and T. Fountain, "Fault rates in nanochip devices," in *Electrochem. Soc. Proc.*, vol. 99, 1999, pp. 354–368.

[34] K. Nikolić, A. Sadek, and M. Forshaw, "Fault-tolerant techniques for nanocomputers," *Nanotechnology*, vol. 13, pp. 357–362, 2002.

[35] J. Han and P. Jonker, "A system architecture solution for unreliable nanoelectronic devices," *IEEE Trans. Nanotechnol.*, vol. 1, pp. 201–208, Dec. 2002.

[36] J. Han and P. Jonker, "A defect- and fault-tolerant architecture for nanocomputers," *Nanotechnology*, vol. 14, pp. 224–230, 2003.

[37] F. J. MacWilliams and N. J. Sloane, *The Theory of Error-Correcting Codes*: NorthHolland, 1978.

[38] D. A. Spielman, "Highly fault-tolerant parallel computation," in *Proc. 37th Annual IEEE Conference on Foundations of Computer Science*, 1996, pp. 154–163.

[39] H. Nishio and Y. Kobuchi, "Fault tolerant cellular spaces," *J. Comput. Syst. Sci.*, vol. 11, pp. 150–170, 1975.

[40] M. Harao and S. Noguchi, "Fault tolerant cellular automata," *J. Comput. Syst. Sci.*, vol. 11, pp. 171–185, 1975.

[41] P. Gács, "Reliable computation with cellular automata," *J. Comput. Syst. Sci.*, vol. 32, no. 1, pp. 15–78, 1986.

[42] ——, "Self-correcting two-dimensional arrays," in *Advances in Computing Research (a Scientific Annual)*, vol. 5, S. Micali, Ed.. Greenwich, CT, 1989, pp. 223–326. Randomness in Computation.

[43] P. Gács and J. Reif, "A simple three-dimensional real-time reliable cellular array," *J. Comput. Syst. Sci.*, vol. 36, no. 2, pp. 125–147, 1988.

[44] W. Wang, "An Asynchronous Two-Dimensional Self-Correcting Cellular Automaton," Ph.D. dissertation, Boston University, Boston, MA 02 215, 1990. Short version: In Proc. 32nd IEEE Symposium on the Foundations of Computer, IEEE Press, pp. 188–192, 1991.

[45] P. Gács, "Reliable cellular automata with self-organization," *J. Stat. Phys.*, vol. 103, no. 1/2, pp. 45–267, 2001. Short version in Proc. IEEE Symposium on Foundations of Computer Science, pp. 90–99, 1997.

[46] T. Isokawa, F. Abo, F. Peper, S. Adachi, J. Lee, N. Matsui, and S. Mashiko, "Fault-tolerant nanocomputers based on asynchronous cellular automata," *Int. J. Mod. Phys.*, 2004, to be published.

[47] J. Lee, F. Peper, S. Adachi, and S. Mashiko, "Universal delay-insensitive systems with buffering lines," *IEEE Trans. Circuits Syst., Part I*, 2003, submitted for publication.

[48] A. W. Burks and J. von Neumann, Eds., *The theory of Self-Reproducing Automata*. Champaign, IL: Univ. Illinois Press, 1966.

[49] E. F. Codd, *Cellular Automata*. New York: Academic, 1968.

[50] K. Morita and S. Ueno, "Computation-universal models of two-dimensional 16-state reversible cellular automata," *IEICE Trans. Inform. Syst.*, vol. E-75-D, no. 1, pp. 141–147, 1992.

[51] N. Margolus, "Physics-like models of computation," *Phys. D*, vol. 10, pp. 81–95, 1984.

[52] T. Toffoli and N. Margolus, "Invertible cellular automata: A review," *Phys. D*, vol. 45, pp. 229–253, 1990.

[53] J. Lee, F. Peper, S. Adachi, K. Morita, and S. Mashiko, "Reversible computation in asynchronous cellular automata," in *Proc. 3rd Int. Conf. Unconventional Models Computation*, C. S. Calude, M. J. Dinneen, and F. Peper, Eds., 2002, pp. 220–229.

[54] A. Davis and S. M. Nowick. (1997) "An Introduction to Asynchronous Circuit Design". Computer Science Dept., Univ. Utah. [Online]. Available: http://www.cs.columbia.edu/async/publications.html

[55] J. Lee, F. Peper, S. Adachi, and S. Mashiko, "Universal delay-insensitive circuits with bidirectional and buffering lines," *IEEE Trans. Comput.*, to be published.

[56] R. M. Keller, "Toward a theory of universal speed-independent modules," *IEEE Trans. Comput.*, vol. C-23, pp. 21–33, 1974.

[57] J. C. Ebergen, "A formal approach to designing delay-insensitive circuits," *Distrib. Comput.*, vol. 5, pp. 107–119, 1991.

[58] P. Patra, "Approaches to the Design of Circuits for Low-Power Computation," Ph.D. Thesis, University of Texas at Austin, 1995.

[59] S. M. Ornstein, M. J. Stucki, and W. A. Clark, "A functional description of macromodules," in *Proc. Spring Joint Computer Conf. (AFIPS)*, 1967, pp. 337–355.

[60] Encyclopedia of Delay-Insensitive Systems (Edis) [Online]. Available: http://edis.win.tue.nl/edis.html

[61] F. Peper, T. Isokawa, F. Abo, J. Lee, S. Adachi, N. Matsui, and S. Mashiko, "Fault-tolerance in biological systems simulated on asynchronous cellular automata," in *Proc. 9th Int. Symp. Artificial Life Robotics (AROB)*, 2004, pp. 61–66.

[62] J. A. Reggia, S. L. Armentrout, H.-H. Chou, and Y. Peng, "Simple systems that exhibit self-directed replication," *Science*, vol. 259, pp. 1282–1287, 1993.

[63] M. Kutrib and R. Vollmar, "Minimal time synchronization in restricted defective cellular automata," *J. Inform. Process. Cybern.*, vol. 3, pp. 179–196, 1991.

[64] M. Kutrib and R. Vollmar, "The firing squad synchronization problem in defective cellular automata," *IEICE Trans. Inform. Syst.*, vol. E78-D, no. 7, pp. 895–900, 1995.

[65] H. Umeo, "A fault-tolerant scheme for optimum-time firing squad synchronization," *Parallel Comput.: Trends Applicat.*, pp. 223–230, 1994.

Ferdinand Peper (M'00) received the M.S. and Ph.D. degrees in computer science from Delft University of Technology, Delft, The Netherlands, in 1985 and 1989, respectively.

Currently, he is a Senior Researcher at the Nanotechnology Group of the Communications Research Laboratory, Kobe, Japan, and Visiting Professor at the Himeji Institute of Technology, Japan. His research interests include cellular automata, nanocomputing, quantum computing, neural computing, and evolutionary hardware.

Jia Lee received the B.E, M.E., and Ph.D. degrees from Hiroshima University, Japan, in 1996, 1998, and 2001, respectively.

He is currently a Postdoctoral Researcher in the Nanotechnology Group of the Communications Research Laboratory, Kobe, Japan. His research interests include asynchronous circuits, asynchronous cellular automata, and formal language theory.

Fukutaro Abo received the B.E. degree from Himeji Institute of Technology in 2001 and is currently working toward the M.E. degree at the same institution.

His research interests include fault- and defect-tolerant cellular automata.

Teijiro Isokawa (S'99–M'01) received the M.E. degree from Himeji Institute of Technology in 1999.

He is currently working as a Research Associate at the Division of Computer Engineering, Graduate School of Engineering, Himeji Institute of Technology. His research interests include cellular automata-based computer systems, evolutionary computation, artificial immune systems, and neural networks.

Susumu Adachi received the M.E. degree in physical engineering from Hiroshima University, Japan, in 1995 and the Ph.D. degree in computer engineering from Kobe University, Japan, in 2001.

He is currently a Postdoctoral Researcher at the Nanotechnology Group of the Communications Research Laboratory, Japan. His research interests include cellular automata, quantum computation, and evolutionary computation.

Nobuyuki Matsui received the B.S. degree in physics from the Faculty of Science, Kyoto University, Japan, in 1975 and the M.E. and Dr. Eng. degrees in nuclear engineering from Kyoto University in 1977 and 1980, respectively.

After working in the Faculty of Science and Technology, Kinki University, he became an Associate Professor and a Professor of the Computer Engineering Division, Graduate School of Engineering at Himeji Institute of Technology in 1993 and 1998, respectively.

Dr. Matsui is a Member of the ENNS, SICE, ISCIE, and the Physical Society of Japan.

Shinro Mashiko received the B.S. degree in 1980 from Yamanashi University, Kofu, Japan, and the M.E. and Dr. degrees from Tohoku University, Sendai, Japan, in 1983 and 1987, respectively.

He is the Chief of the Nanotechnology Section and the Director of the Kansai Advanced Research Center in Kobe, Japan, which is part of the Communication Research Laboratory, Japan.

Dr. Mashiko is a Member of the Japan Society of Applied Physics.