Appl. Phys. A 80, 1183–1195 (2005)
DOI: 10.1007/s00339-004-3154-4

Applied Physics A
Materials Science & Processing

Nanoelectronic architectures

G. SNIDER✉
P. KUEKES
T. HOGG
R. STANLEY WILLIAMS

Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304, USA

Received: 13 September 2004/Accepted: 23 November 2004
Published online: 11 March 2005 • © Springer-Verlag 2005

ABSTRACT Configurable crossbars are the easiest computational structures to fabricate at the nanoscale. By creating multiple types of crossbars and assembling them into larger structures, we may implement general computation. Architectures for diode-based and transistor-based logic are presented, along with latching mechanisms. We present simulation results from defect-tolerance studies on two applications (a 3-bit adder and a 4-bit microprocessor) mapped onto defective, nanoelectronic fabrics, and outline strategies for fault tolerance.

PACS 85.35.-p; 85.40.Bh; 85.40.Qx; 85.65.+h

1 Introduction

Nanoelectronics offers the potential for much denser circuitry than is possible with current CMOS technology, but presents a number of challenges that must be addressed if we are to successfully exploit it. At the size scales we are considering (< 15 nm), conventional lithography will lack the resolution necessary to create the individual devices normally combined to create larger circuits. Sensitivity to noise, subatomic particles, and even quantum uncertainty at this scale will increase the rate of transient faults to the point that redundancy and fault tolerance will become necessary for logic functions as well as memory. And, regardless of fabrication technique, nanoscale circuits will likely contain defects so numerous that it would be uneconomical to simply discard circuits containing a single defect – some form of defect tolerance will be necessary to achieve acceptable yields.

These challenges have led us to focus on configurable crossbar architectures [6, 7, 11, 15, 20, 25–27, 33, 51], where each cross point within a crossbar can be independently configured to activate an electronic device, such as a resistor, diode, or transistor. Crossbars are one of the easiest structures to build using nanoimprint lithography [6, 23, 24]. Their high degree of redundancy offers a simple strategy for defect tolerance. Their regular structure makes them easy to analyze for devising a strategy for fault tolerance. Configurability offers the potential for using a single nanoelectronic fabric for a large number of applications, much like field-programmable gate arrays, reducing design costs.

The architectures we describe here are necessarily speculative to varying degrees since many of the crossbar devices we hypothesize are either not yet functional in the laboratory, or are functional but incompletely characterized. Configurable nanoscale transistor crossbars, for example, do not yet exist; consequently we use simplified, idealized models for them.

2 Crossbars, interlayers, and junctions

The word crossbar denotes both an interconnection topology and a fabrication geometry. A nanoelectronic crossbar consists of two or more parallel planes of nanoscale wire arrays (‘nanowires’), with each pair of planes separated by a thin layer of a chemical species (called the ‘interlayer’) with particular electrochemical properties (see Fig. 1). The region where a wire in one plane crosses over a wire in the other plane is called a ‘junction’. The nature of the interlayer and the type of wires used determine the type of device formed at each junction (Fig. 2).

Some junctions have the desirable property of being configurable, meaning that, at some time after manufacturing, the devices at selected junctions may be independently activated or deactivated [8]. An activated device (such as a resistor) functions as a normal electrical component; a deactivated device appears to have functionally vanished. A configurable resistor junction, for example, behaves as if it were a resistor in series with a switch that may be opened or closed. When closed, the resistor connects the upper and lower wires in the junction; when open, the resistor has no effect on the cir-

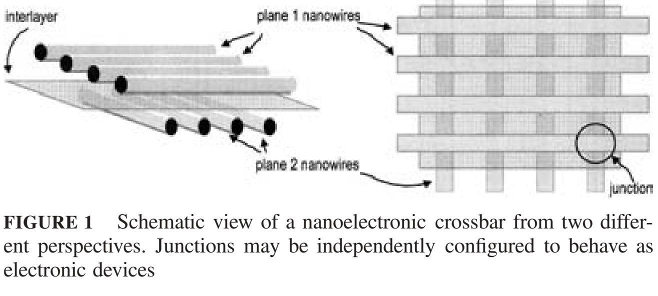


FIGURE 1 Schematic view of a nanoelectronic crossbar from two different perspectives. Junctions may be independently configured to behave as electronic devices

✉ Fax: +1-650-813-3312, E-mail: snider.greg@hp.com

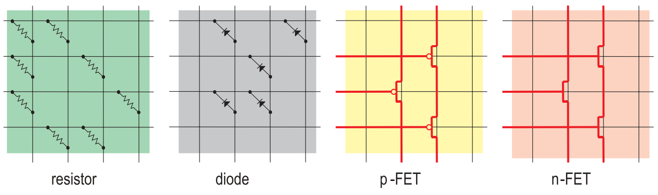


FIGURE 2 Tiles. Depending on the nature of the nanowires and interlayer, different electronic devices may be configured at the junctions, although only a single device type is available for each tile. Normally a junction is 'non-functional' in that there is no interaction between the two wires that define it; when an appropriate voltage differential is applied to the two wires, the electronic component is configured, e.g. a switch is closed that connects the wires through the device

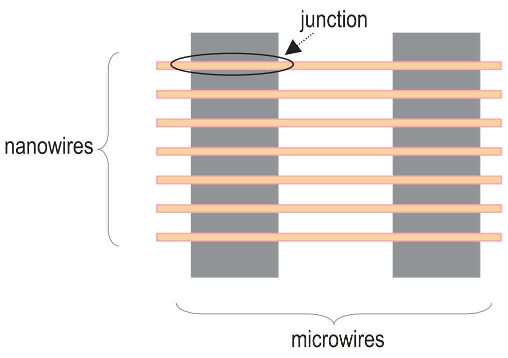


FIGURE 3 A tile formed by molecular wires ('nanowires') crossing submicron wires ('microwires') separated by an interlayer. These junctions may also be configured to form different electronic components

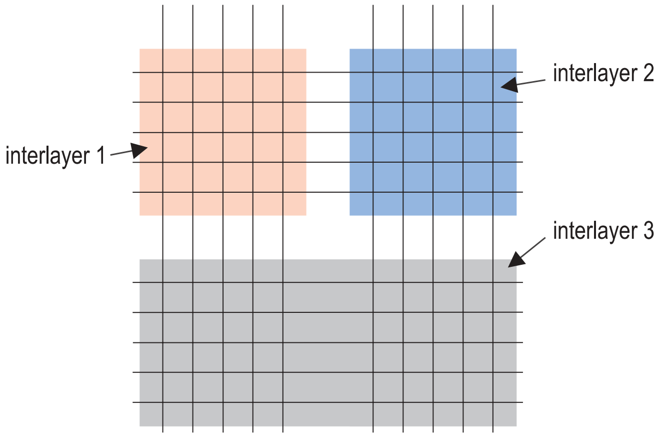


FIGURE 4 A 'mosaic' of tiles is created by using different interlayers in different regions of the crossbar

cuit. Configuring the device is the act of setting the switch to be open or closed. This may be done electrically, optically, mechanically, or by some other means. Typically, a junction in a crossbar that we fabricate is configured electrically by applying different voltages to the two wires forming the junction. A device which is reconfigurable may be repeatedly activated and deactivated.

Junctions may also be formed between a nanowire and a submicron-scale wire (also called a 'microwire') on an underlying substrate, such as conventional CMOS (Fig. 3). Such junctions may also be configured to form electronic components, depending upon the composition of the wires and the separating interlayer.

2.1 *Tiles*

Submicron-scale masking can be used to realize multiple device types in side-by-side regions within a single crossbar such that the nanowires maintain electrical conductivity across and between regions. We refer to a single-device-type region as a tile, our fundamental architectural building block. Several adjacent tiles can be combined into larger functional units, which are referred to as composite tiles or blocks. Thus, in Fig. 4, the junctions in the gray interlayer region may be configured to form closed cross-point switches, while junctions in the pink and blue regions may be configured to form *n*-FETs or *p*-FETs, respectively [16, 22, 29]. Alignment precision for masking, as compared to nanowire pitch, limits the minimum tile size to roughly 10×10 junction regions of the crossbar.

A single tile is insufficient to implement logic, but composite tiles or blocks can create many different families of logic depending upon the tiles that comprise them. For example, Fig. 5 schematically illustrates a logic block implementation of an AND/OR gate using diode/resistor logic. The horizontal and vertical lines in that figure represent nanowires, with the horizontal wires in one plane and the

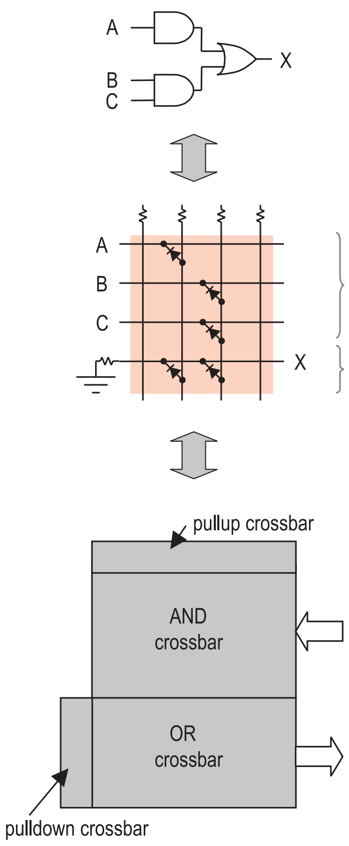


FIGURE 5 A logic block implementing AB+C gate using diode/resistor logic

vertical wires in another. Pullup and pulldown crossbars are configurable resistor tiles between nanowires (horizontal and vertical lines) and submicron-scale microwires on a substrate that supply power and ground. Selected junctions in these tiles are configured to supply the pull-up and pull-down resistors required by diode logic. A configurable diode tile is used to implement the diodes required for the AND and OR gates.

The diode/resistor logic block in Fig. 5 is, perhaps, the simplest to fabricate, but cannot implement logical inversion or signal regeneration. A more powerful logic block based on *n*-FETs and *p*-FETS, known as the 'complementary/symmetry array' [50], is shown in Fig. 6. This particular logic block is capable of implementing AND/OR/INVERT gates, powerful enough for general computation (Fig. 7). Note that it is possible to design such complex gates so that they map easily onto the crossbar logic block. Details may be found in [47].

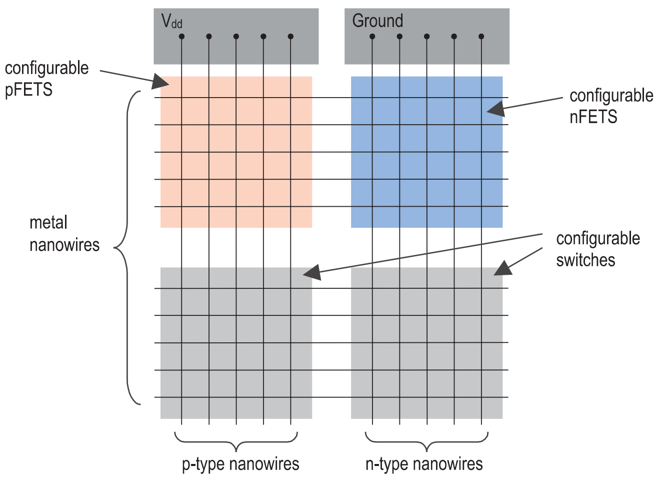
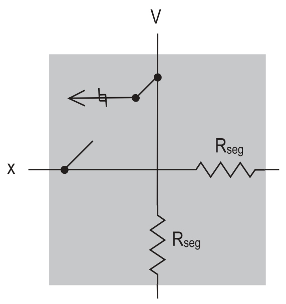


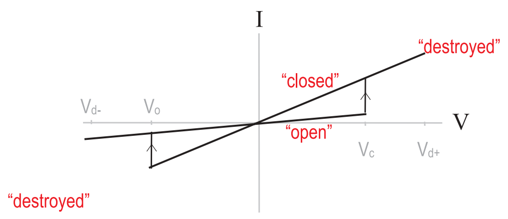
FIGURE 6 A logic block implemented with the complementary/symmetry array. Each junction in the pink quadrant may be independently configured to implement a *p*-FET, while each junction in the blue quadrant may be configured to implement an *n*-FET. The junctions in the two lower quadrants may be configured to be 'closed' switches, representing a low-impedance path between the two nanowires defining the junction

2.2 *Latches*

Diode/resistor logic, such as shown in Fig. 5, requires signal restoration to regenerate signals degraded by voltage drops across diodes and resistors within the logic. Without such restoration, it is impossible to implement basic memory devices like latches and flip-flops. We can augment the diode/resistor logic block with a basic latch block that implements both signal restoration and memory. The latch block is based on a 'hysteretic resistor' junction, shown in Fig. 8.



a



b

V_{d-}	-3.0 V
V_o	-1.75 V
V_c	1.25 V
V_{d+}	2.0 V
R_{open}	1 Gigohm
R_{closed}	1 Megohm
R_{seg}	100 ohms

c

FIGURE 8 The 'hysteretic resistor' junction. Conceptually this is a switch across a junction that can be opened (high-impedance state) or closed (low-impedance state), shown schematically in (**a**). An idealized I/V curve for such a device is shown in (**b**), with typical parameters listed in (**c**)

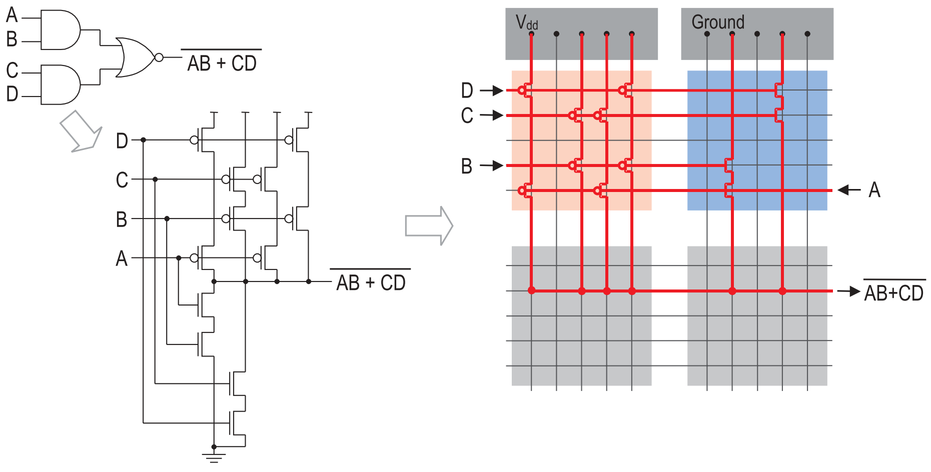


FIGURE 7 Implementing logic by selectively configuring junctions in the complementary/symmetry array

The hysteretic resistor junction is normally in one of two states: high impedance ('open') or low impedance ('closed') [6, 52]. The switch remains in whatever state it is in as long as the voltage drop across it (measured at 'V' relative to 'x' in Fig. 8a) remains in the operating range $[V_o, V_c]$. In the open state, though, the switch will transition to the closed state if the voltage drop across the switch exceeds V_c; in the closed state, the switch will transition to the open state if the voltage drop is less than V_o. However, an excessive positive or negative voltage drop across the junction will destroy it.

An array of one-bit latches can be built from two hysteretic resistor tiles as shown in Fig. 9. This block consists of a number of nanowires crossing two microwires, with each microwire forming a tile with the nanowires that cross it. Note that the two side-by-side tiles possess switches of opposite polarity; the configuration voltages applied to activate junctions along the ControlA microwire are of the opposite polarity for junctions along ControlB. It is for this reason that two different $1 \times N$ crossbars (tiles) are needed rather than a single $2 \times N$ crossbar. Input signals enter the nanowires on the left and a sequence of voltages applied to the microwires causes the value of the input signals to be stored in the junctions of microwires 'ControlA' and 'ControlB', effectively creating a latch for each of the nanowires. Details of the operation of this latch are sketched in Fig. 10 and described in detail by Kuekes [28]. This latch not only stores input data values, it also regenerates the signal voltage and may deliver the output signals in either true or inverted form. These properties make it suitable for use with diode/resistor logic (which lacks the inversion and signal-regeneration capabilities) to create synchronous circuits capable of universal computation. Note that, since the input and output to each latch share the same nanowire, additional circuitry is required to provide isolation between the input and output at appropriate times.

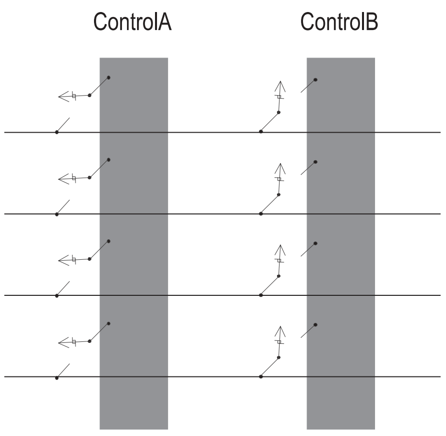


FIGURE 9 This latch array stores data for each signal entering a nanowire on the *left* and drives the stored signals out to the *right*

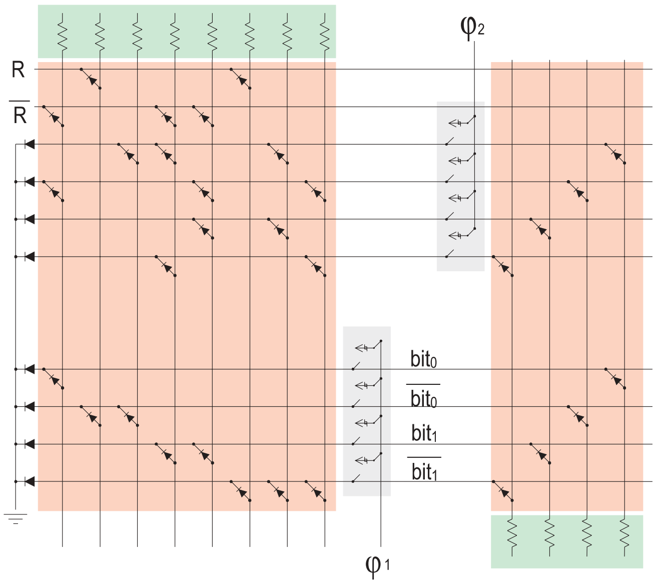


FIGURE 11 A two-bit counter implemented with impedance-encoded latches and an unusual form of diode/resistor logic

This latch uses conventional voltage encoding at its output. A simpler latch can be constructed using only one of the tiles shown in Fig. 9, but the resulting latch uses impedance rather than voltage to encode the stored logic value. Combining this latch with an unusual form of diode/resistor logic enables one to build sequential logic circuits, forming a foundation for general computation. Figure 11 shows an example of this logic/latch family – a resettable, two-bit counter. Details are given in [49].

2.3 *Mosaics and fabrics*

Tiles and blocks may be composed to form even larger structures, called 'mosaics', that perform some useful function. One example of a mosaic is the antisymmetric array shown in Fig. 12, which takes advantage of circuit locality (often referred to as 'Rent's rule' [31]) to provide a low-overhead interconnect between circuit components that are strongly connected with each other.

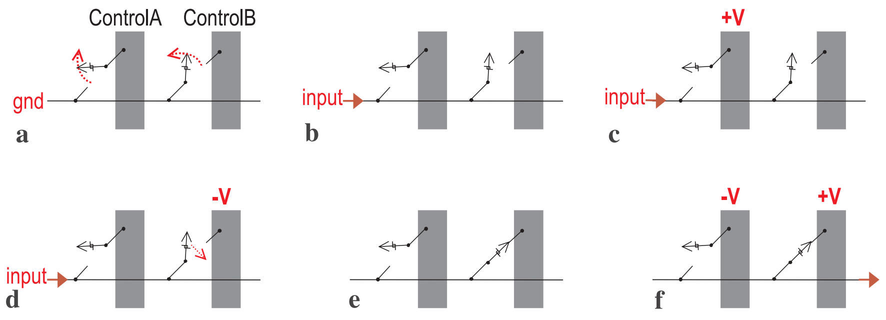


FIGURE 10 Operation of a single latch within the array. (**a**) Both switches are unconditionally opened by application of appropriate voltages to ControlA and ControlB; (**b**) an input signal is applied to the horizontal nanowire; (**c**) ControlA is driven with a positive voltage that will cause the switch to close only if the input is 'low'; (**d**) ControlB is driven with a negative voltage that will cause its switch to close only if the input is 'high'; (**e**) the input signal is removed from the input – at this point exactly one of the two switches will be closed; (**f**) the latch outputs the held signal by driving the two control microwires

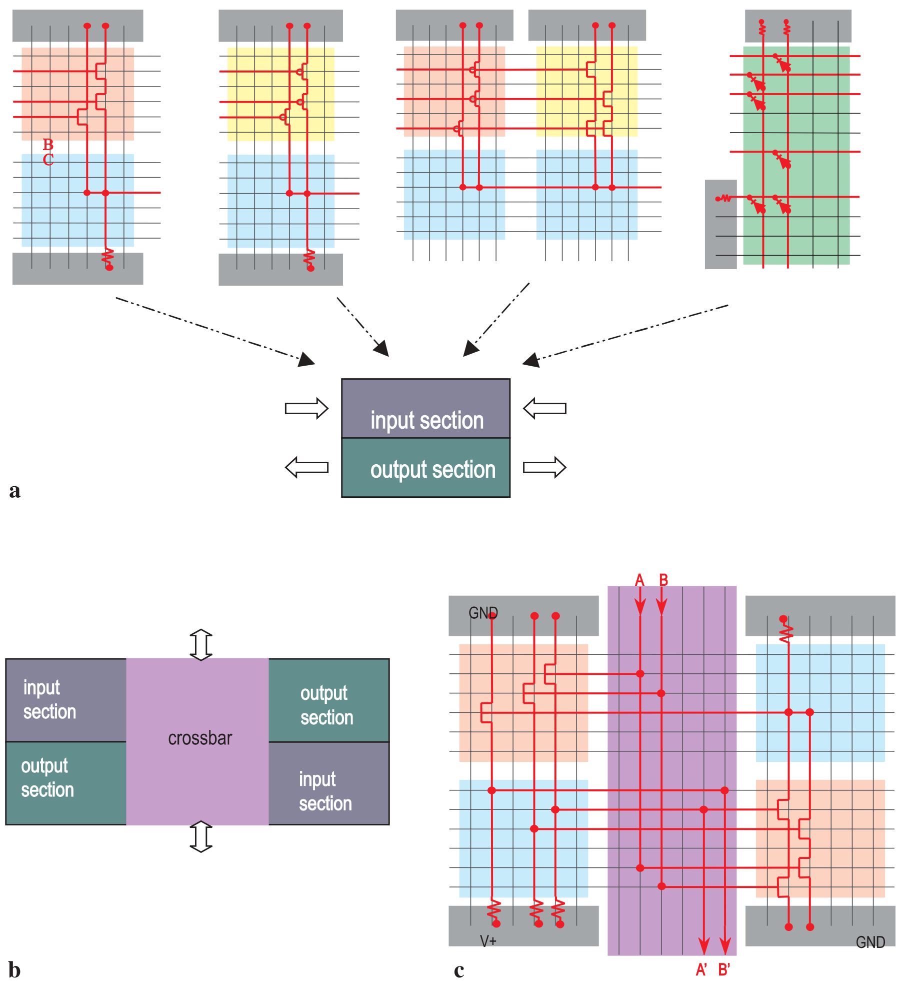


FIGURE 12 Antisymmetric arrays are constructed from logic blocks (of almost any logic family) to create a 'supermosaic'. The large structure takes advantage of the locality of connection implied by 'Rent's rule', providing low-overhead connectivity between logical components that are closely interacting. (**a**) Logic block structure; (**b**) antisymmetric array, built out of two logic blocks (one flipped upside down) and a crossbar. Signals between the two logic blocks are exchanged without using any of the vertical wire resources of the central crossbar; (**c**) modulo-4 incrementer implemented using the antisymmetric array: [B'A']=[BA]+1

Using reflections and rotations, two complementary/symmetry arrays (Fig. 6) can be interleaved to form the configurable bidirectional buffer block shown in Fig. 13. This block can take a signal on any wire on either side, amplify it (with inversion), and drive it out on any wire on the opposite side. The choice of direction on each wire and the mapping of an input wire to an output wire are independent, making this a useful component in building a routing network, and for 'stitching together' nanowires from two different regions to effectively form longer nanowires.

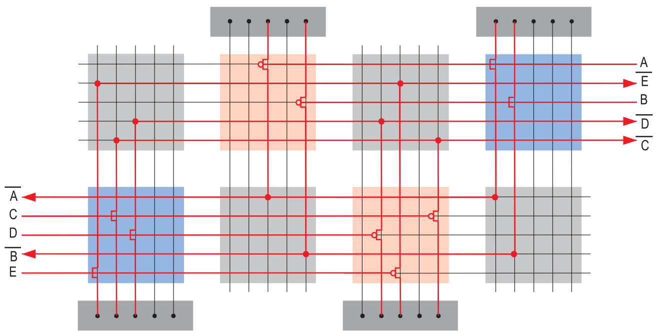


FIGURE 13 Bidirectional buffer array. This structure regenerates each signal without introducing a delay from latching. Each nanowire may be independently configured for direction

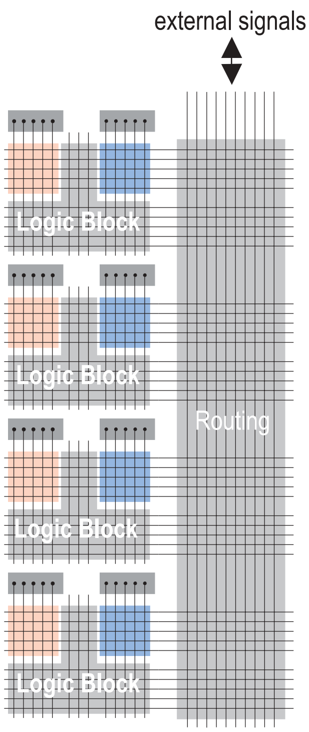


FIGURE 14 Logic fabric: logic and routing resources combined to form a larger computational structure

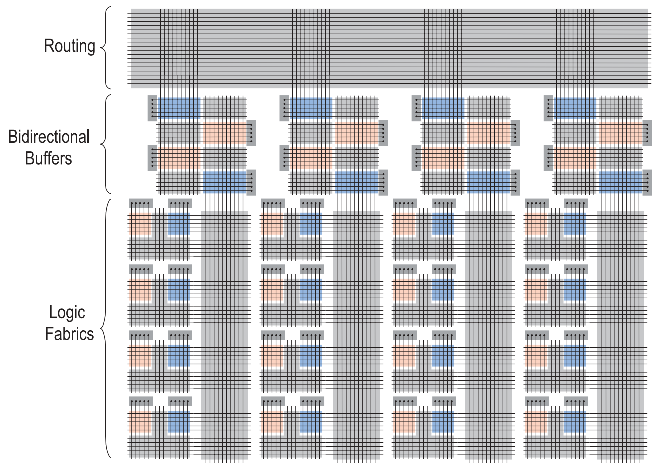


FIGURE 15 Larger computing fabrics may be built out of combinations of logic and routing blocks, hierarchically organizing them in a fractal-like pattern

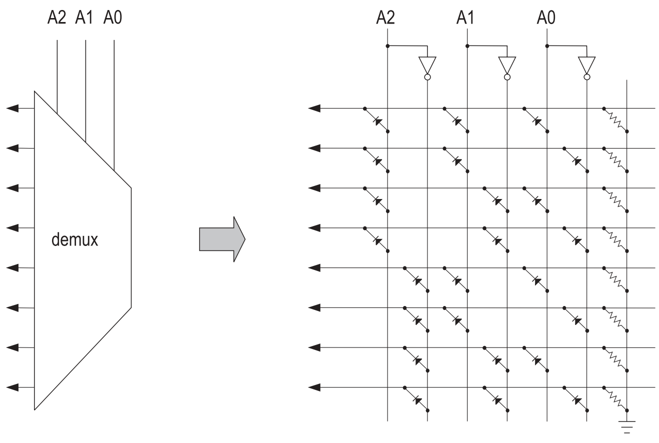


FIGURE 16 A demultiplexer, shown schematically on the *left*, selects one of N output lines as directed by an input address [A2 A1 A0]; an implementation of a demultiplexer, shown on the *right*, using a diode crossbar and resistors

Larger computational structures can be built by combining the arrays with configurable switch tiles for routing. One example is the logic fabric shown in Fig. 14. The routing structure on the right provides local communication between the logic arrays on the left as well as communication with external signals. One can hierarchically combine structures, such as the logic block, bidirectional buffer block, and routing tiles, to build still larger fabrics as shown in Fig. 15.

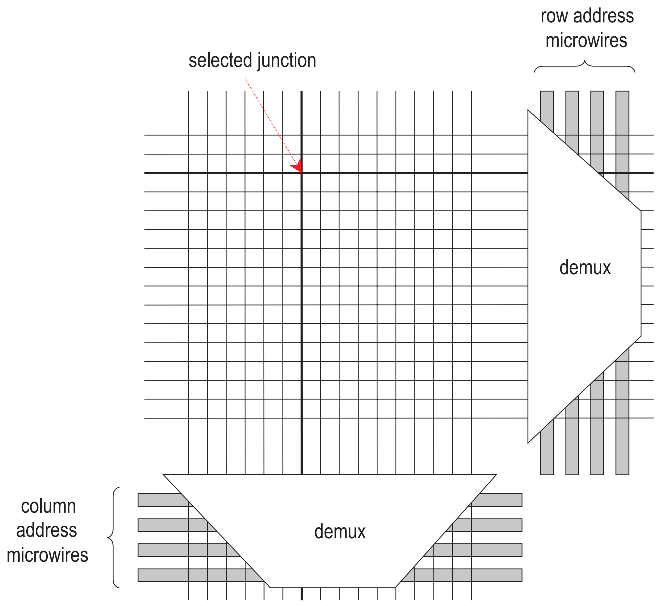


FIGURE 17 Using a row demultiplexer and a column demultiplexer allows for the selection of a single junction with a crossbar

3 Configuration and the micro/nanointerface

Nanoelectronic circuits must interface with the outside world, both for input/output and to allow the nanocrossbars to be configured. Although one can envision, for example, photonic or other interfaces, the initial interfaces we are pursuing are with conventional CMOS. This leads to a problem: the scale and pitch of the nanowire crossbars are incompatible with the comparably coarse features of CMOS. We need a strategy for interfacing the submicron world with the nanoworld.

Demultiplexers are an appealing mechanism for this interface [26, 58]. They allow a small number of submicron-scale wires (microwires) to control a larger number of nanoscale wires (nanowires) by forcing a 'selected' voltage onto exactly one of the nanowires, and a different, 'unselected', voltage onto the rest. This provides a mechanism for a CMOS circuit on a silicon substrate to interrogate and configure a nanoelectronic 'chip' residing on its surface. Demultiplexers also have the appealing property that they can be efficiently implemented in crossbars.

Figure 16 illustrates a demultiplexer implementation using a diode crossbar with specific junctions configured. The address lines of the demultiplexer are microwires that could be implemented on a CMOS substrate; the output data lines are nanowires crossing the microwire address lines, forming a nanowire/microwire crossbar. If we have full control over which junctions are configured in the crossbar, we need only about $2 \log N$ address microwires to control N nanowires. If we lack such control, but have a fabrication process that allows each junction to be independently configured with probability 0.5, the number of address wires increases to about $5 \log N$ [26].

Placing two demultiplexers along adjacent edges of a nanowire/nanowire crossbar gives us a means to 'select' exactly one junction within that crossbar. As shown in Fig. 17, the demultiplexer driving the horizontal nanowires (rows) might be addressed to select one of those wires by driving it with a positive voltage (for simplicity, we assume that unselected outputs are driven with ground); the demultiplexer driving the vertical nanowires (columns) might be addressed to drive a selected column with a negative voltage (again, unselected outputs are assumed to be driven to ground). The result is that the junction defined by the intersection of the driven column and the driven row has a voltage drop across it that is larger than any other junction in the crossbar. This allows us to configure (and deconfigure) junctions sensitive to voltage drops across them.

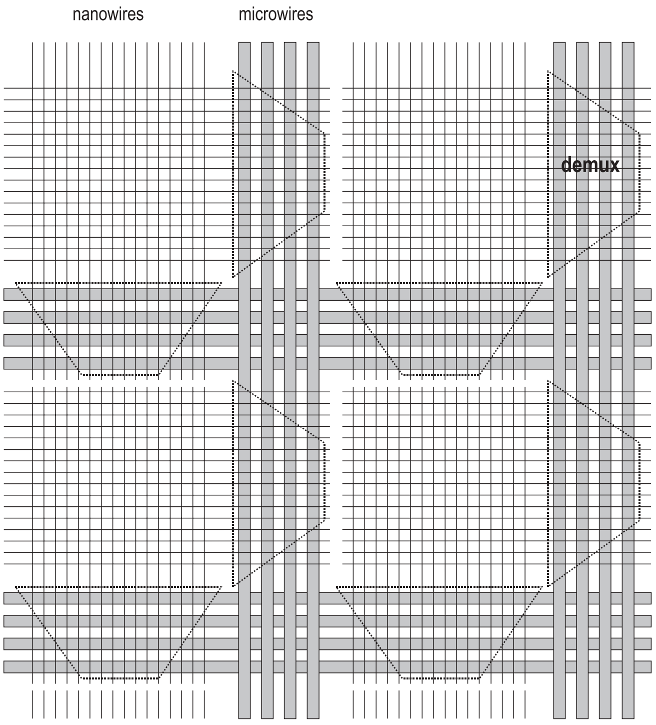


FIGURE 18 Microwires carrying address lines may be shared among multiple demultiplexers

Figure 18 illustrates how microwires can be shared among several demultiplexers in order to configure multiple crossbars.

4 Defect tolerance

For nanoscale crossbars, the main type of defect is that introduced during manufacture (so-called 'static defects') rather than during operation. This is reasonable for plausible technologies, which involve high temperatures during manufacture, and hence a relative ease of introducing defects, but low temperature during operation, with much less chance of creating new defects. In this situation, an appropriate systems architecture requires a compiler to arrange for desired circuit behaviors by using only correctly functioning junctions within crossbars, as determined by a testing phase after manufacture [9, 19, 34]. This approach of avoiding known defects yields a defect-tolerant architecture. It contrasts with methods dealing with defect-induced faults that appear during operation, perhaps intermittently, e.g. using majority votes from replicated hardware.

For simplicity, we restrict our attention here to defects leading to 'unconfigurable' junctions, rather than defects that short out a junction or adjacent wires, or that unexpectedly cause a break in a wire. In this scenario, we can test the circuits to determine which junctions are defective, and then use the remaining ones to implement the circuit. That is, a compiler takes the required logic function and a table of defects to find a way to implement that function on the defective crossbar fabric.

This leads to the central question addressed in this section: given a defect rate and a certain-size crossbar (or crossbar fabric), how likely is it that we can find a way to implement a particular logic function in that crossbar? Determining whether such a circuit exists, and finding it if it does, is a combinatorial search problem. Thus, a related problem is the computational difficulty for the compiler to identify an implementation, or conclude that no implementation is possible. Decreasing the allowable defect rate for a nanoelectronic fabric will generally require more difficult and costly manufacturing. On the other hand, increasing the allowable defect rate will make it less likely that a desired circuit can be implemented, and can also result in longer run times for the compiler to identify a way to implement the circuit while avoiding defects.

A further aspect of this problem is that logic functions can often be written in different but logically equivalent forms. For example, (a OR b) AND c is logically equivalent to (a AND c) OR (b AND c). These rewrites can involve different numbers of terms and, it turns out, affect the compiler's ability to successfully map the function onto a defective fabric.

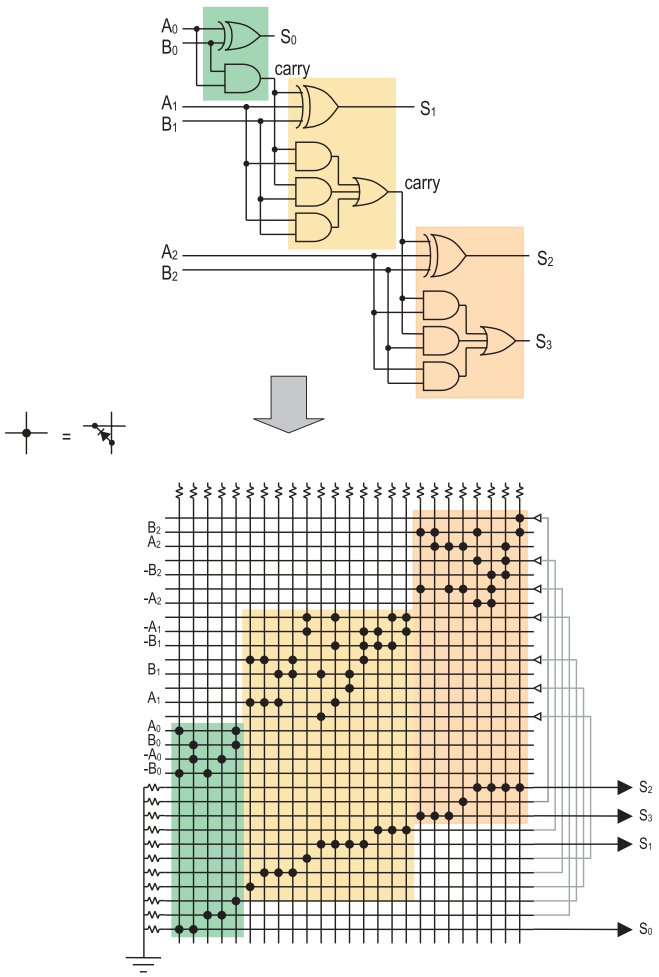


FIGURE 19 A 3-bit adder which adds two 3-bit numbers (denoted as the bits $A_2A_1A_0$ and $B_2B_1B_0$) to produce a 4-bit sum ($S_3S_2S_1S_0$). The ripple-carry implementation (*top*) translates directly to a diode crossbar implementation (*bottom*) using feedback from some of the outputs to the inputs (*gray lines*). Regenerative buffers (*left-pointing triangles*) between stages regenerate signals degraded by diode and resistor voltage drops. The input wire marked $-A_0$ gives the complement of input bit A_0, and similarly for the other inputs. Note that the carry bit between successive stages of the crossbar implementation must be presented in both original and complemented forms

4.1 *3-bit adder*

We first consider the mapping of a small, 3-bit adder circuit onto a diode/resistor logic block (as was illustrated in Fig. 5). There are several different ways to implement this circuit. Figure 19 shows a straightforward 3-bit, ripple-carry adder that is essentially a direct translation of the logic circuit shown at the top, producing four output bits, $S_0 \ldots S_3$, representing the sum of the two 3-bit numbers. Because this implementation uses several levels of logic, some of the intermediate output signals must be fed back to some of the inputs, possibly requiring signal regeneration in the process to compensate for degradation due to diode and resistor voltage drops.

A second implementation of the 3-bit adder is shown in Fig. 20. Here the entire circuit uses only two logic levels, eliminates the need for feedback, and requires less area. On the other hand, it requires more diode junctions and uses a greater number of diodes along some of the vertical and horizontal wires. For instance, the circuit in Fig. 19 never uses more than four diodes on any wire, while the circuit of Fig. 20 requires as many as 16. Thus we might expect that this circuit, which requires packing more diodes into a smaller area, might be more difficult to implement on a defective crossbar.

To examine the behavior of implementing adder circuits on defective crossbars, we created a number of simulated test cases. Specifically, for a given adder implementation (e.g. single- or multiple-stage) and crossbar size, we mark each junction of the diode crossbar as defective independently with probability p. Because the pull-up and pull-down resistor junctions would have much lower defect rates (since they are formed by microwire/nanowire junctions, of considerably larger area than the diode nanowire/nanowire junctions), we restrict our attention to cases with no defective resistors. A compiler was used to automatically map the adder onto the defective crossbar. A compilation was characterized by three parameters: (1) defect probability, p; (2) adder implementation (single- or multiple-stage); (3) crossbar area, a (number of junctions in the diode crossbar). An experiment consisted of 50 attempts to compile a single implementation onto 50 different crossbars with defect probability p and area a.

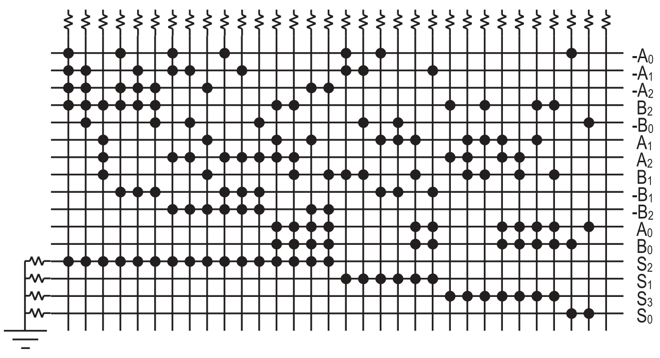


FIGURE 20 A 3-bit adder implemented as 2-level logic in a single diode crossbar. Although this approach uses more diodes, it consumes less area, avoids the feedback and regenerative buffers between stages, and will likely offer less propagation delay. Inputs and outputs are labeled as in Fig. 20 (the right-most column wire is not used in this circuit)

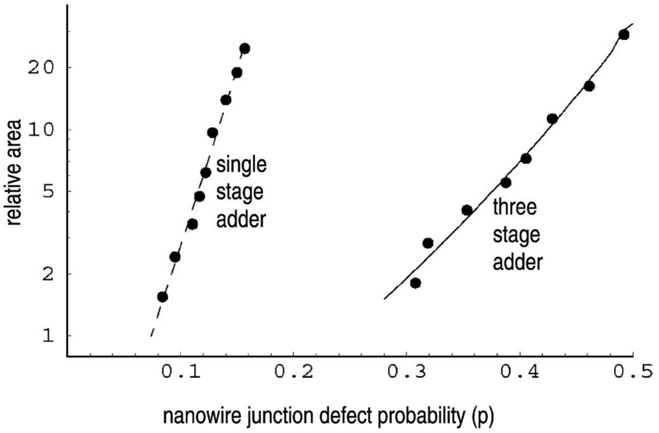


FIGURE 21 Relative area required to have at least 90% probability of being able to find a correctly functioning 3-bit adder circuit as a function of defect probability, p. Areas are relative to that required for a single-stage adder on a defect-free crossbar

Figure 21 shows the results from our experiments of implementing a 3-bit adder onto fixed-area crossbars. We see a threshold behavior where P_{circuit}, the probability of being able to successfully map the adder onto the defective crossbar, drops abruptly over a fairly short range of p values. The 3-stage adder can tolerate higher defect rates than the single-stage implementation. More details can be found in [21].

4.2 *4-bit microprocessor*

Next we consider the mapping of a more complex application, a 4-bit microprocessor, onto a defective crossbar fabric. The target fabric consisted of 64 identical logic blocks implementing a variant of the complementary/symmetry array (Fig. 6) combined with a routing network, creating a structure similar to that shown in Fig. 15. The routing network contained abundant routing resources to ensure that any failure to map the application onto the fabric was caused by an inability to allocate resources due to defects rather than simple routing congestion. Defects were again limited to the 'stuck-open' type, randomly distributed throughout all junctions formed by the crossing of two nanowires. Junctions formed by a nanowire crossing a microwire were assumed to be non-defective.

The 4-bit microprocessor was implemented in 143 lines of C code. The instruction memory for the processor contained a short program for implementing a two-pole, low-pass filter using 18 words of six bits; this memory was included in the compiled circuit. Figure 22 shows an example of the compilation of the microprocessor onto the crossbar fabric. In this particular case, 6% of all junctions have been marked defective (red x's). The compiler routes around the defects, implementing logic functions by allocating non-defective junctions (yellow dots) and wires.

As was the case with the 3-bit adder, the manner in which individual logic functions are synthesized affects the ability of the compiler to successfully map onto the fabric. For this application, we varied the maximum number of input variables, m, that logic synthesis was allowed in creating a sum-of-product representation of a logic function. For example, AND gates with up to m inputs could implemented, while AND

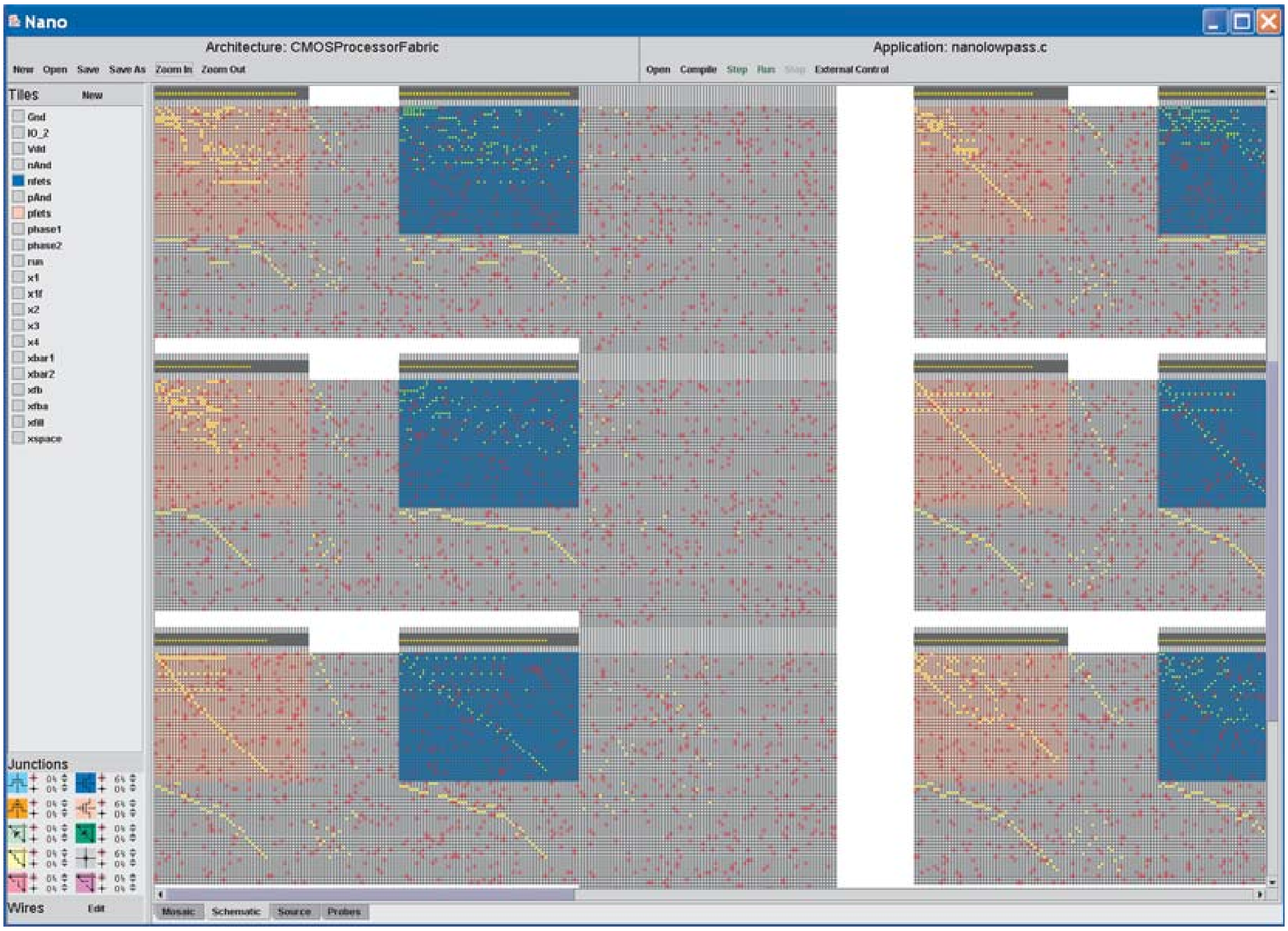

Nano
Architecture: CMOSProcessorFabric
New Open Save Save As Zoom In Zoom Out
Application: nanolowpass.c
Open Compile Step Run Stop External Control
Tiles New
Gnd
IO_2
Vdd
nAnd
nfets
pAnd
pfets
phase1
phase2
run
x1
x1f
x2
x3
x4
xbar1
xbar2
xfb
xfba
xfill
xspace
Junctions
Wires Edit
Mosaic Schematic Source Probes

FIGURE 22 A portion of the experimental fabric configured with the microprocessor application. Junctions marked with red 'x's are assumed defective; junctions with yellow dots are those which have been configured by the compiler

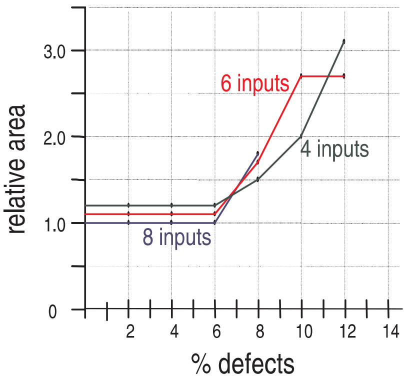


FIGURE 23 Area required for the 4-bit microprocessor as a function of defect rate and the synthesis parameter inputs. The defect rate is the probability that a given nanowire/nanowire junction will be non-functional ('stuck open'). The inputs parameter is the maximum number of inputs allowed on an AND gate – increasing inputs creates smaller, denser circuitry that is more difficult to map onto defective fabrics

gates with $m+1$ inputs could not. Larger values of m produce denser logic implementations, but these are also more difficult to map onto defective crossbars.

Figure 23 summarizes experiments compiling the microprocessor onto the fabric with varying defect rates and using different values of the mapping parameter, m. We find that small values of m yield circuits that are indeed easier to map onto defective crossbars, but the lower density requires more crossbar area to contain the less-dense circuit. Conversely, higher values of m yield circuits that are denser and much harder to map. The trade-off in selecting a value of m depends upon the expected defect rate. For more details, see [47].

5 Fault tolerance

We distinguish faults (transient perturbations of the circuit that could cause it to function incorrectly) from defects (permanent flaws in the nanofabric). Faults, as we have defined them, are also referred to as single-event upsets (SEUs) or soft errors. They can be caused by subatomic particles striking a conductor within a circuit, briefly altering the charge, and hence the voltage, in that conductor. Nanoelectronic structures will be more susceptible to faults than conventional CMOS [38] since

(1) Fewer electrons will be used to represent logic states in nanologic circuits, increasing their sensitivity to changes in charge.
(2) Devices will likely have a much greater variability in device parameters, reducing voltage safety margins.
(3) Quantum probabilities will become more visible at the nanoscale, so that devices may not behave as reliably or predictably as one would like.

Faults can cause both logic and memory devices to function incorrectly, a fatal situation for sequential machines – once a bit error occurs, is stored and fed back as part of the computation, the final result will be, with very high probability, incorrect. There are some computations where this is not the case: infinite impulse response (IIR) digital filters, for example, would tend to reconverge to a more-or-less correct output given sufficient time following a fault. But most computations are not so forgiving, and one must devise methods for detecting faults when they occur and then somehow correcting or masking them.

Circuits that continue correct operation in the presence of random faults below some fault rate threshold are said to be fault tolerant. All strategies for fault-tolerant design use redundancy, either by replicating or augmenting circuitry (spatial redundancy [35]), repeating a computation (temporal redundancy [2, 37]), or a combination of the two.

5.1 *Reliable logic*

Spatial redundancy involves rewriting logic circuits such that internal variables (consisting of one or more bits in the original circuit) are replaced with wider bit representations chosen from a particular redundant code. Such a code is carefully constructed so that all likely faults will cause an affected variable to take on a bit pattern that is not a legal member of the code. The presence of a non-codeword in an internal variable is thus a signal that a fault has occurred, and that error correction is required before the computation can be continued.

The simplest such code is a replication code, which simply repeats a bit value one or more times – if all replicants do not agree on the value of the single bit in the original circuit that they collectively represent, an error has occurred. Error correction in that case can be implemented with ‘voting’, using a majority rule to select the ‘correct’ bit state, or with more subtle schemes.

In one of the earliest papers addressing logic and faults [53], von Neumann proposed two schemes for using such replication codes. The first, called N-modular redundancy, requires N copies of a circuit (where N is an odd integer greater than two) to execute in parallel, with the correct output determined by majority voting. This approach is only effective if the voter is much more reliable than the replicated circuits, and is therefore used chiefly at the system level where, for example, N large computers execute the identical computation in parallel while a small (and therefore more reliable) voter checks their outputs for agreement. This is the approach taken in life-critical space-flight applications [14]. Note that this will only work if the probability of more than $\lfloor N/2 \rfloor$ systems failing within the same voting interval is lower than some acceptable threshold.

Von Neumann’s second approach, which has been called ‘parallel restitution’ by one author [45], works at the opposite end of the circuit spectrum, using bit-level redundancy to detect and correct errors at the gate level. This strategy replaces each internal bit variable with a ‘bundle’ of bits that collectively represent the original bit, and then replicates the gate driving the original bit to the same degree. Ideally all bits within the bundle would have the same logic value – faults, though, will cause some of the bits to disagree with the others. Instead of voting, von Neumann combined simple gates with complex wiring in a clever way to create ‘stochastic amplifiers’ that tend to reduce the disagreement within a bundle without completely eliminating it. He was able to show that such a scheme could maintain the coherence of a computation, effectively correcting errors in place. The drawback is that the required degree of replication can be extremely high, thousands or more, depending on the error rate, making the scheme impractical.

Between the system level and the gate level lie opportunities for more efficient codes than simple replication. Addition, for example, is so highly structured that there exist error-correcting codes that are much more efficient than replication codes. Demultiplexing is another example of where this is possible [48]. Hadjicostis has generalized this to observe that efficient codes can be developed for group and semigroup operations [17]. Other approaches between the system and gate levels include microarchitectural [44], register transfer level, and algorithm level recomputation [56].

Since error detection is generally easier than error correction, much work has been devoted to schemes (collectively called concurrent error detection or CED [35]) that aim to ensure data integrity of logic systems. The goal is to construct redundant circuits (called totally self-checking circuits [30, 54]) that are self testing and fault secure, meaning that they either produce correct results or indicate that their output is incorrect. For replication codes, this implies that a mere doubling of subsystems might be sufficient (as opposed to the triplication or worse required for majority voting schemes) – if two copies of a subsystem produce results that disagree, an error has occurred. The duplication can be identical, but it has been shown that diverse duplication – two different implementations of the same logic function – is more robust against multiple failures [36]. Non-replication codes for concurrent error detection include parity prediction [1], Berger codes [3], and Bose–Lin codes [10].

Figure 24 shows the basic CED approach for separable codes, where check bits are generated from the input bits and concatenated with a logic function’s output bits to create

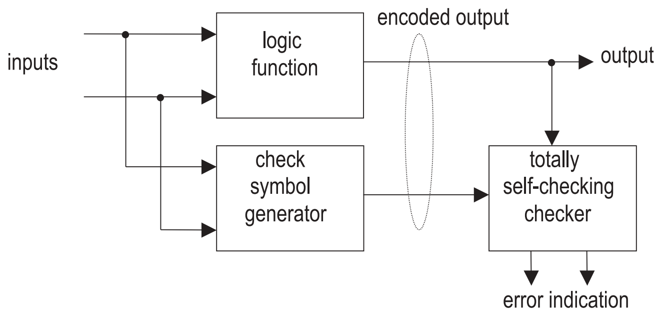


FIGURE 24 A concurrent error detection (CED) scheme using a separable code. Inputs to a logic function produce the normal logic output, but additional circuity (the check-symbol generator) produces check bits that are concatenated with the normal output bits to form an encoded result. This result is then checked by a totally self-checking checker to verify that it is a legal codeword. The output of the checker is itself encoded, usually using a 2-bit, 1-of-2 code, so that faults within the checker are detected as well. For example, an output of (1, 0) might signify a correct result, while any other bit combination would flag an error in the logic function, check symbol generator, or totally self-checking checker

a coded output word. Separable codes have the desirable property of trivial decoding, since the actual data is just a subset of the encoded bits. This encoded output is checked with a 'totally self-checking checker' to verify that the output is, in fact, a legal codeword. The output of the checker must be encoded with a redundant code to catch faults within the checker itself. A concrete example of this using a parity-prediction code is shown in Fig. 25. The top of the figure shows a 3-bit incrementer implemented in diode and resistor crossbars – it takes a 3-bit input and outputs the sum of that input and one, modulo 8. A fault-tolerant version is shown at the bottom of Fig. 25, using the scheme of Fig. 24. The 3-bit output of the incrementer is augmented with a single parity bit, *p*, produced by the parity-predictor circuit to produce a 4-bit encoded result. The 4-bit output is a legal codeword only if it contains an even number of 1's, known as even parity. The totally self-checking checker is composed of two parts, a parity generator, which predicts the parity of the three output data bits, and a totally self-checking equality checker, which compares the generated parity bit with the predicted bit (they should be the same). (Note that this is logically equivalent to checking the combined four bits for even parity, but results in a smaller circuit.) The output of the equality checker is a two-bit code: an output of (1, 0) implies a correct result, while (0, 1) implies a fault in either the incrementer or the predictor; an output of (1, 1) or (0, 0) implies a fault in the equality checker. This encoding presumes that no more than one fault occurs at a given time anywhere in the circuit.

Temporal redundancy approaches such as alternate-data retry [46], recomputation with shifted operands [40], or delay-line/latch comparison [37] trade off space and time. Generally, these approaches produce fault-tolerant circuits that are smaller than spatially redundant circuits, at the cost of greater execution time.

5.2 *Reliable state machines*

Building reliable state machines (computations that require some internal variables to be stored and fed back to the inputs) is much more challenging than reliable logic [12, 13, 32, 39, 41, 57]. To start with, latches used to hold state variables must be 'SEU (single event upset) hardened' [4, 5, 42, 43, 55] so that state data is not altered once stored. The latches must also be designed to work with the redundant logic feeding its inputs, otherwise a glitch on a latch input could result in the capturing of invalid data – it does no good to reliably store a wrong answer.

One structure often implicitly used in fault-tolerant state machines is the Muller C element or C gate [18]. The C gate, used extensively in asynchronous circuit design, is a kind of 'voting memory element' that has two or more inputs: the output of the gate remains the same until all of its inputs change to a new value, at which point the output becomes the agreed upon new input value. C gates therefore inherently filter out glitches on inputs as long as not all inputs glitch at the same time. C gates on the outputs of duplicated subsys-

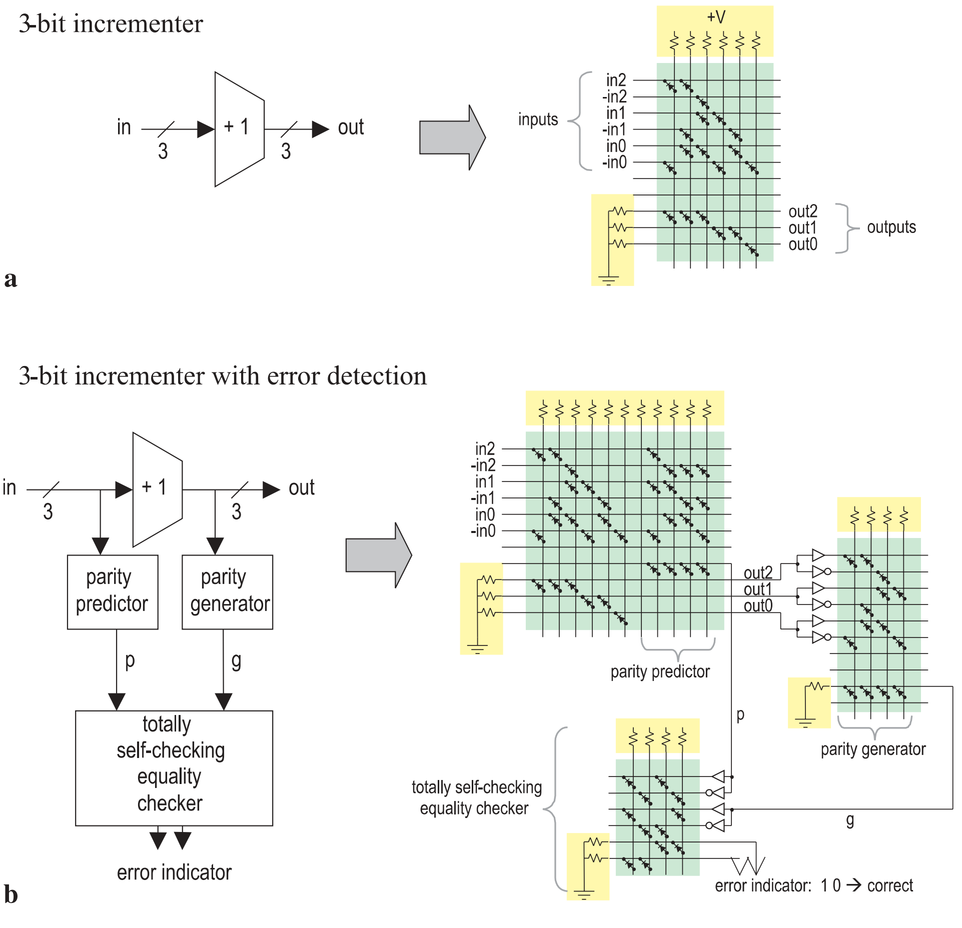


FIGURE 25 Adding concurrent error detection to a 3-bit incrementer circuit. The incrementer computes (input+1) mod 8. The *top figure* (**a**) shows a crossbar implementation using diode/resistor logic. The *bottom figure* (**b**) augments the implementation to detect errors using the scheme shown in Fig. 24. There is a considerable increase in circuit size, and additional buffers and inverters (whose implementations are not shown) are required

tems thus provide a kind of voting system where the output of the gate does not change unless both input subsystems agree that a change is desired. Of course the output of a C gate can glitch due to a subatomic particle strike as well, so it is necessary that the C gate be fused into the latch design [2, 37]. Even with SEU-hardened latches, though, it can still be desirable to use redundant codes for representing state variables. Anghel's dynamic C gate design [2] is easily mapped onto the complementary/symmetry array (Fig. 26).

One interesting paradigm for building reliable nanoelectronic state machines is 'checkpoint/restart': the correctness of the state machine is checked at each cycle using one or more concurrent error detection schemes, and computation is allowed to continue only if no error is detected. If an error occurs, the state machine is 'rolled back' to the beginning of the cycle and the computation is retried. This makes the execution time of a computation non-deterministic, but provides a graceful degradation as fault rates increase – the computation simply slows down. Of course there will exist a threshold fault rate, above which the circuit will simply be unable to make much forward progress. The number of retries is typically limited to some maximum value in order to detect either excessive fault rates or a post-manufacturing-induced defect (which can be handled by reconfiguration).

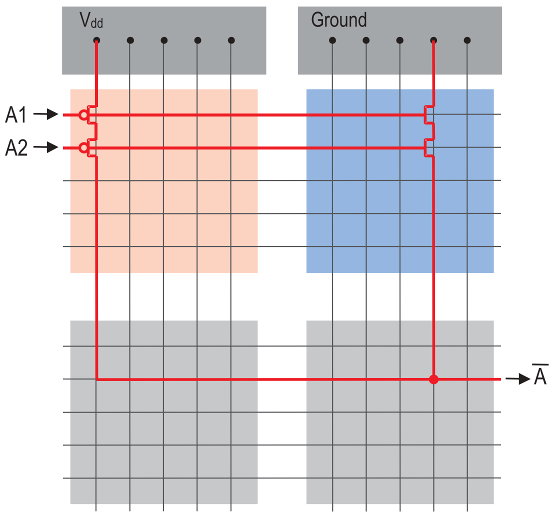


FIGURE 26 A dynamic implementation of an inverting C gate in the complementary/symmetry array. If both inputs, A1 and A2, agree, then the output will be equal to their complement. If either of the two inputs 'glitches', the output will be undriven for the duration of the glitch. Assuming sufficient capacitance on the output, and assuming that the glitch duration is not too long, the output will hold its state, effectively filtering out the glitch

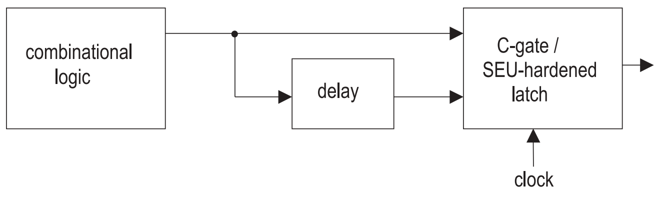


FIGURE 27 A delay line preceding a C-gate-based latch creates a 'glitch' filter that can prevent transient errors in the output of a logic function from being captured in the latch. The delay must be longer than the longest duration expected for any single-event upset

Nicolaidis's time-redundancy scheme [37], sketched in Fig. 27, is a promising approach that avoids logic replication or additional coding circuitry. Here the output of a logic function is copied and fed to a delay line. The original output and delayed output are then fed to the inputs of a C gate. As long as the delay introduced by the delay line is longer than the maximum length of a transient pulse due to a single-event upset, the C gate will act as a 'glitch filter', preventing erroneous transients from being captured in the latch.

It is unlikely that a single fault-tolerance scheme will be sufficient for all applications being mapped onto a nanoelectronic substrate. There are scenarios where *N*-modular redundancy would be most appropriate because of its simplicity and non-invasive approach, and other situations (such as systolic computations) where one can capitalize on either the algorithmic structure or efficient coding to reduce the overhead of fault tolerance when this is economically desirable. It is partly for this reason that we have chosen a compilation approach to nanoelectronic architecture: a compiler can automate the process of analyzing the application to be mapped to the nanohardware, select the most appropriate fault-tolerance strategy (under constraints supplied by the designer), and automatically implement the sometimes difficult and intrusive transformations needed to achieve fault-tolerance goals.

6 Conclusion

Shrinking electronics to the nanoscale will bring huge challenges that can be met by intelligent architecture – simplified structures to make manufacturing easier, reconfigurability and redundancy to make circuit implementation possible in the presence of defects, and appropriate strategies implemented by a compiler to deal with run-time faults.

ACKNOWLEDGEMENTS We thank Institute of Physics Publishing for granting permission to reproduce several figures (1, 4, 6, 7, 13, 14, 15, 22, and 23) from an earlier paper [47]. Some of the research performed in the Quantum Science Research group of HP labs is supported by the Defense Advance Projects Agency (DARPA).

REFERENCES

1 S. Almukhaizim, Y. Makris: 'Fault Tolerant Design of Combinational and Sequential Logic Based on a Parity Check Code'. In: *Proc. 18th IEEE Int. Symp. Defect and Fault Tolerance in VLSI Systems, 3–5 November 2003*, Boston, MA, USA, pp. 563–570
2 L. Anghel, D. Alexandrescu, M. Nicolaidis: 'Evaluation of a Soft Error Tolerance Technique Based on Time and/or Space Redundancy'. In: *Proc. 13th Symp. Integrated Circuits and Systems Design, 18–24 September 2000*, Manaus, Brazil, pp. 237–242
3 J. Berger: Inf. Control **4**, 68 (1961)
4 D. Bessot, R. Velazco: 'Design of SEU-hardened CMOS Memory Cells: the HIT Cell'. In: *Second Eur. Conf. Radiation and its Effects on Components and Systems (RADECS 93), 13–16 September 1993*, Saint-Malo, France, pp. 563–570
5 T. Calin, M. Nicolaidis, R. Velazco: IEEE Trans. Nucl. Sci. **43**, 2874 (1996)
6 Y. Chen, G. Jung, D. Ohlberg, X. Li, D. Stewart, J. Jeppesen, K. Nielsen, J. Stoddart, R. Williams: Nanotechnology **14**, 462 (2003)
7 Y. Chen, R.S. Williams: Configurable nanoscale crossbar electronic circuits made by electrochemical reaction, US Patent No. 6 518 156 (2003)
8 C. Collier, E. Wong, M. Belohradsky, F. Raymo, J. Stoddart, P. Kuekes, R. Williams, J. Heath: Science **285**, 391 (1999)
9 B. Culbertson, R. Amerson, R. Carter, P. Kuekes, G. Snider: 'Defect Tolerance on the Teramac Custom Computer'. In: *Proc. 1997 IEEE Symp.*

FPGA's from Custom Computing Machines (FCCM '97), Napa Valley, CA, USA, pp. 116–123

10 D. Das, N. Touba: J. Electron. Test.: Theory Appl. (JETTA), **15**, 145 (1999)

11 A. DeHon: IEEE Trans. Nanotechnol. **2**, 23 (2003)

12 P. Drineas, Y. Makris: 'Non-Intrusive Design of Concurrently Self-Testable FSMs'. In: *Proc. 11th Asian Test Symp. (ATS'02)*, Guam, USA, 18–20 November 2002, pp. 33–38

13 P. Drineas, Y. Makris: 'Non-Intrusive Concurrent Error Detection in FSMs through State/Output Compaction and Monitoring via Parity Trees'. In: *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE '03)*, Messe Munich, Germany, 3–7 March 2003, pp. 1164–1165

14 J. Gaisler: 'Concurrent Error-detection and Modular Fault Tolerance in a 32-bit Processing Core for Embedded Space Flight Applications'. In: *Proc. 24th Annu. Int. Symp. Fault-Tolerant Computing*, Austin, Texas, 15–17 June 1994, pp. 128–130

15 S. Goldstein, M. Budiu: 'NanoFabrics: Spatial Computing Using Molecular Electronics'. In: *Proc. 28th Int. Symp. Computer Architecture, ISCA, 2001*, 30 June–4 July 2001, Goteborg, Sweden, pp. 178–191

16 L.J. Guo, P.R. Krauss, S.Y. Chou: Appl. Phys. Lett. **71**, 1881 (1997)

17 C. Hadjicostis: *Coding Approaches to Fault Tolerance in Combinational and Dynamic Systems* (Kluwer Academic Publishers, Boston, Dordrecht, London 2002)

18 S. Hauck: Proc. IEEE **83**, 69 (1995)

19 J. Heath, P. Kuekes, G. Snider, R.S. Williams: Science **280**, 1716 (1998)

20 J. Heath, M. Ratner: Phys. Today, May, 43 (2003)

21 T. Hogg, G. Snider: Defect-tolerant logic with nanoscale crossbar circuits, submitted to IEEE Trans. Nanotechnol.

22 Y. Huang, X.F. Duan, Y. Cui, L.J. Lauhon, K.H. Kim, C.M. Lieber: Science **294**, 1313 (2001)

23 G.Y. Jung, S. Ganapathiappan, D.A.A. Ohlberg, D.L. Olynick, Y. Chen, W.M. Tong, R.S. Williams: Appl. Phys. **78**, 1169 (2004)

24 G.Y. Jung, S. Ganapathiappan, D.A.A. Ohlberg, D.L. Olynick, Y. Chen, W.M. Tong, R.S. Williams: Nano Lett. **4**, 1225 (2004)

25 P.J. Kuekes, R.S. Williams, J.R. Heath: Molecular wire crossbar memory, US Patent No. 6 128 214 (2000)

26 P. Kuekes, R.S. Williams: Demultiplexer for a molecular wire crossbar network, US Patent No. 6 256 767 (2001)

27 P.J. Kuekes, R.S. Williams, J.R. Heath: Molecular-wire crossbar interconnect (MWCI) for signal routing and communications, US Patent No. 6 314 019 (2001)

28 P. Kuekes: Molecular crossbar latch, US Patent No. 6 586 965 (2003)

29 P.J. Kuekes, R.S. Williams: Molecular wire transistor (MWT), US Patent No. 6 559 468 (2003)

30 P. Lala: *Fault Tolerant and Fault Testable Hardware Design* (Prentice-Hall, London 1985)

31 B. Landman, R. Russo: IEEE Trans. Comput. **20**, 1469 (1971)

32 C. Metra, S. Di Francescantonio, M. Omana: 'Automatic Modification of Sequential Circuits for Self-Checking Implementation'. In: *Proc. 18th IEEE Int. Symp. Defect and Fault Tolerance in VLSI Systems (DFT '03)*, 03–05 November 2003, Boston, MA, USA

33 Y. Luo, C. Collier, J. Jeppesen, K. Nielsen, E. Delonno, G. Ho, J. Perkins, H. Tseng, T. Yamamoto, J. Stoddart, J. Heath: Chem. Phys. Chem. **3**, 519 (2002)

34 M. Mishra, S. Goldstein: 'Defect Tolerance at the End of the Roadmap'. In: *Proc. Int. Test Conf. 2003 (ITC 2003)*, 30 September–2 October 2003, Vol. 1, Charlotte, NC, USA, pp. 1201–1210

35 S. Mitra, E. McCluskey: 'Which Concurrent Error Detection Scheme to Choose?'. In: *Proc. Int. Test Conf. 2000*, 3–5 October 2000, Atlantic City, NJ, USA, pp. 985–994

36 S. Mitra, E. McCluskey: 'Design Diversity for Concurrent Error Detection in Sequential Logic Circuits'. In: *Proc. 19th IEEE VLSI Test Symp. (VTS 2001)*, 29 April–3 May 2001, Marina Del Rey, CA, USA, pp. 178–183

37 M. Nicolaidis: 'Time Redundancy Based Soft-error Tolerance to Rescue Nanometer Technologies'. In: *Proc. 17th IEEE VLSI Test Symp.*, 25–29 April 1999, San Diego, CA, USA pp. 86–94

38 P. Packen: Science **24**, 2079 (1999)

39 R. Parekhji, G. Venkatesh, S. Sherlekar: 'Concurrent Error Detection Using Monitoring Machines'. In: *IEEE Design and Test of Computers, Fall 1995*, Vol. 12, No. 3, pp. 24–32

40 J. Patel, L. Fung: IEEE Trans. Comput. C **31**, 589 (1982)

41 S. Piestrak: 'Limitations of Design Methods for Self-Checking Synchronous Sequential Machines'. *FastAbstract 29th Int. Symp. Fault-Tolerant Computing*, Madison, Wisconsin, 15–18 June 1999, Session 6C: Fast Abstracts I, Madison, WI, USA

42 L. Rockett: Science **35**, 1682 (1988)

43 L. Rockett: Single-event upset hardened reconfigurable bi-stable CMOS latch, US Patent No. 6 369 630 (2002)

44 E. Rotenberg: 'AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors'. In: *Proc. 29th Fault-Tolerant Computing Symposium*, 15–18 June 1999, Madison, WI, USA, pp. 84–91

45 A. Sadek, K. Nikolic, M. Forshaw: Nanotechnology **15**, 192 (2004)

46 J. Shedletsky: IEEE Trans. Comput., February, 106 (1978)

47 G. Snider, P. Kuekes, R.S. Williams: Nanotechnology **15**, 881 (2004)

48 G. Snider, W. Robinette: Crossbar demultiplexers for nanoelectronics based on N-hot codes, to appear in IEEE Trans. Nanotechnol.

49 G. Snider, P. Kuekes: Nano state machines using hysteretic resistors and diode crossbars, in preparation

50 G. Snider, P. Kuekes, R.S. Williams: Configurable molecular switch array, US Patent Application Publication No. US 2004/0041617 A1, 4 March 2004

51 M. Stan, P. Franzon, S. Goldstein, J. Lach, M. Ziegler: Proc. IEEE, November, 1940 (2003)

52 D.R. Stewart, D.A.A. Ohlberg, P.A. Beck, Y. Chen, R.S. Williams, J.O. Jeppesen, K.A. Nielson, J.F. Stoddart: Nano Lett. **4**, 133 (2004)

53 J. von Neumann: 'Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components'. In: *Automata Studies* (Princeton University Press 1956) pp. 43–98

54 J. Wakerly: *Error Detecting Codes, Self-Checking Circuits and Applications* (Elsevier/North-Holland, New York 1978)

55 S. Whitaker, J. Canaris, K. Liu: Nucl. Sci. **38**, 1471 (1991)

56 K. Wu, R. Karri: IEEE Trans. Computer-Aided Design Integr. Circuits Syst. **21**(9), 1077 (2002)

57 C. Zeng, N. Saxena, E. McCluskey: 'Finite State Machine Synthesis with Concurrent Error Detection'. In: *Proc. ITC Int. Test Conf.*, 27–30 September 1999, Atlantic City, NJ, USA, pp. 672–679

58 Z. Zhong, D. Wang, Y. Cui, M. Bockrath, C. Lieber: Science **302**, 137 (2003)