Chapter 29
Self-Awareness in Digital Systems: Augmenting Self-Modification with Introspection to Create Adaptive, Responsive Circuitry

Nicholas J. Macias and Lisa J.K. Durbeck

Abstract The question of augmenting self-modification with introspection to create flexible, responsive digital circuitry is discussed. A specific self-configurable architecture—the Cell Matrix—is introduced, and features that support introspection and self-modification are described. Specific circuits and mechanisms that utilize these features are discussed, and sample applications that make use of these capabilities are presented. Conclusions are presented, along with comments about future work.

29.1 Introduction

> *"Your visions will become clear only when you can look into your own heart. Who looks outside, dreams; who looks inside, awakes."* [6]

The relevance of Dr. Jung's quote to the human experience seems clear: one must be self-aware to achieve their full potential to "awake" rather than merely to dream. But what might this mean in different contexts: say, in the context of human-made systems?

Self-awareness has been defined by Stephen Franzoi as "…a psychological state in which one takes oneself as an object of attention" [5]. Christopher Jamison further differentiates this from simple introspection, stating "Introspection is only looking at me, whereas self-awareness involves considering how I interact with the world around me" [7]. Under these definitions, it may be interesting to ask if human-made systems can be, in some sense, self-aware.

N.J. Macias (✉)
Clark College, Vancouver, WA, USA
e-mail: nmacias@clark.edu

L.J.K. Durbeck
Cell Matrix Corporation, Newport, VA, USA
e-mail: lisa@cellmatrix.com

© Springer International Publishing Switzerland 2017

A. Adamatzky (ed.), *Advances in Unconventional Computing*,
Emergence, Complexity and Computation 22,
DOI 10.1007/978-3-319-33924-5_29

Looking at digital electronics, one may ask if such circuitry has the necessary mechanisms for any form of self-awareness. In analyzing this question, it may be useful to classify digital circuits as falling into five different classes.

The first class (Class I systems) contains pure digital logic: collections of logic gates, wired together to perform certain operations. In this class, digital circuits can perform algorithmic computations: simple state machines exhibit this behavior.

Class II digital circuitry is circuitry whose structure can be changed. For example, fuse-programmed PLAs have a given initial configuration, but their behavior can be modified (once) by a programming operation that changes the connectivity of the internal elements [1].

The third class extends the notion of configurable hardware to *re-configurable* hardware: devices whose wiring can be modified multiple times. A field-programmable gate array (FPGA) exemplifies this class of hardware systems [8]. In this class, circuitry can be tailored to a particular situation. For example, in a software-defined radio (SDR) system, the hardware which encodes/decodes an audio signal according to one standard can be re-wired to change that coding as the device moves to a different geographic region (where a different coding standard is used) [3]. While Class III circuitry is in some ways more powerful than circuitry from Classes I and II, it requires an additional component, typically a CPU/memory system (e.g. a PC) to generate configuration information for modifying the device. This works well in many situations, but the PC which is generating the configuration strings is fundamentally separate from the hardware that's being modified (Fig. 29.1). In this sense, the PC is not "taking itself as an object of attention," nor is the FPGA. Rather, the PC is considering the FPGA. This misses Franzoi's criteria for self-awareness.

The fourth class of digital circuitry adds the ability of circuitry to *self-modify*. An example of this would be an FPGA containing a multitude of programmable planes, each configured for a different situation, and a small supervisor which chooses which plane to use based on sensory input. Continuing the example above, circuits for different types of cellphone codings could be stored in the device, and a monitor could analyze incoming signals, determining the coding, and selecting the proper

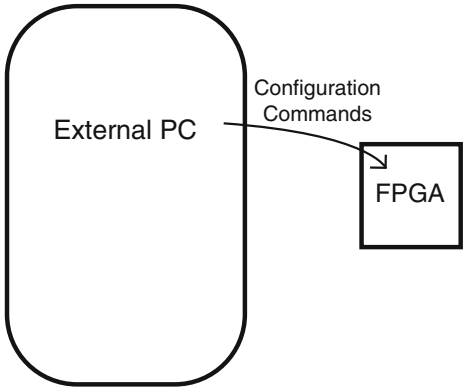

Fig. 29.1 FPGA being configured by an external device. Here the source and object of the configuration are fundamentally different from each other

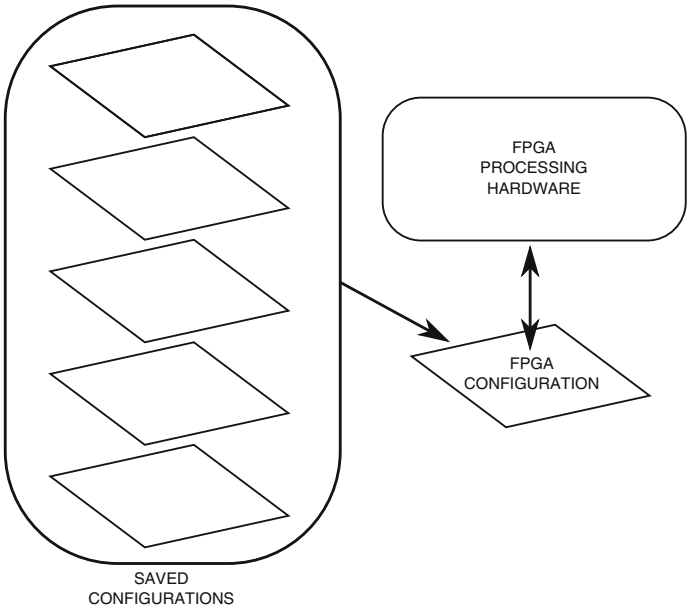

Fig. 29.2 A multi-plane FPGA. Several configurations can be developed and saved in a configuration memory, and then swapped into the actual configuration memory being used by the FPGA

hardware configuration plane (Fig. 29.2). In Class IV, the device is (in some sense) considering itself as an object of attention, since it effectively chooses how to change its digital circuitry based on the results of other pieces of digital circuitry inside itself. This situation is thus arguably one step closer to self-awareness. There is still a key element missing though: the ability to introspect, in the sense of analyzing oneself and making decisions that haven't been pre-wired. Choosing one of several pre-configured circuits based on which one of several possible results comes from an analysis algorithm doesn't "feel" the same as performing a general analysis and synthesizing new circuitry on-the-fly.

With Class V digital circuitry, introspection becomes a central element in the behavior of digital circuits. Circuitry processes information about its own configuration in the same way that it processes other data, and the results of that processing can be used to change or create new digital circuits as easily as changing an output from one value to another. Circuitry in this class is thus able to consider itself as an object of attention, including analyzing how its own circuitry is interacting with other connected circuits, and making decisions about changing its own configuration in response.

This paper discusses Class V circuitry, and describes specific examples of how this combination of introspection and self-modification fosters the implementation of unique and powerful circuits.

29.2 Background

The two key characteristics of Class V reconfigurable devices are:

1. the ability of circuits to *analyze* other circuits configured within the device; and
2. the ability of circuits to *modify* other circuits configured within the device.

These two features mean that the system can analyze and modify itself. The system used for this research is the Cell Matrix [14]. The Cell Matrix was developed in 1986, in part as a way to explore these two aspects of digital circuit design. One of the original goals was to build-in self-modification as an intrinsic piece of the architecture. By also allowing circuits to analyze other circuits, it was hoped that systems could be designed that would, for example, read from a library of sub-circuits and synthesize circuits elsewhere in the matrix by placing and connecting those sub-circuits. A second requirement for the system was that it should be extremely fine-grained: composed of a collection of simple *cells* that, while limited in their individual behavior, could be assembled into arbitrarily-complex circuits. In support of this idea, the third requirement of the system was a high degree of scalability: in particular, connecting two matrices together on their edges should result in a larger matrix, without requiring any changes to either of the pre-existing matrices.

Figure 29.3 shows a simplified view of a single cell within a Cell Matrix. This cell is a 4-sided device, with each side having a single input and a single output. The device is purely combinatorial: the values of the four outputs are determined completely by the contents of a 64-bit *truth table* memory, as shown in Table 29.1.

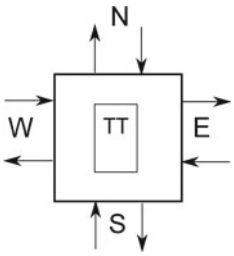

Fig. 29.3 Simplified view of a Cell Matrix cell. The four outputs are determined by the four inputs, which specify a single row in the 16 × 4 truth table ("TT")

Table 29.1 Truth table for a simplified 4-sided cell

N_{in}	S_{in}	W_{in}	E_{in}	N_{out}	S_{out}	W_{out}	E_{out}
0	0	0	0	D_3	D_2	D_1	D_0
0	0	0	1	D_7	D_6	D_5	D_4
0	0	1	0	D_{11}	D_{10}	D_9	D_8
0	0	1	1	D_{15}	D_{14}	D_{13}	D_{12}
…	…						
1	1	1	1	D_{63}	D_{62}	D_{61}	D_{60}

Each of the four outputs are completely determined by the combination of the four inputs

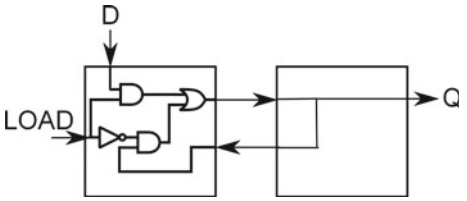

Fig. 29.4 D flip-flop implemented with two cells. The cell on the *left* echoes its eastern input ($LOAD = 0$) or copies its western input (D) to its eastern output ($LOAD = 1$). The eastern cell (on the *right*) re-circulates its western input to its western output, while also copying that input to its eastern output (Q)

By loading the appropriate pattern of 1s and 0s into the truth table memory, a cell can be configured to perform any four combinatorial functions of four inputs (i.e., wires, 1-bit adder, 2-1 selector, basic logic gates, etc.). The combination of 64 bits $(D_0, D_1, D_2, \ldots, D_{63})$ thus completely defines the behavior of a cell. Collections of cells are used to implement more complex function, including those with a sense of state. For example, Fig. 29.4 shows a D flip-flop implemented with two cells. When $LOAD = 1$ the D input is fed into the cell on the right, whereas when $LOAD = 0$ the output from the cell on the right is recirculated back by the cell on the left.

The cell shown in Fig. 29.3 does not include a mechanism for loading bits into the cell's truth table. The addition of a second set of lines—the C (or "control") lines—adds this capability. Figure 29.5 shows a complete cell with these extra lines included. Each side now has two inputs (C and D) and two outputs (C and D). The truth table doubles in size, as shown in Table 29.2.

Now, the combination of 128 bits $(D_0, D_1, D_2, \ldots, D_{127})$ completely specifies the behavior of the cell. While the cell has 8 outputs, those output values are still

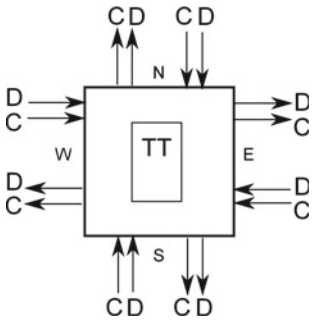

Fig. 29.5 A more-complete view of a Cell Matrix cell, showing both C and D inputs and outputs. The D inputs are used to select a row from the cell's 128-bit (16×8) truth table, in order to generate its 8 outputs. The C inputs are used to place the cell in configuration mode ("C mode"), wherein its truth table can be read or written by a neighbor

Table 29.2 Truth table for a complete 4-sided cell

DN_{in}	DS_{in}	DW_{in}	DE_{in}	CN_{out}	CS_{out}	CW_{out}	CE_{out}	DN_{out}	DS_{out}	DW_{out}	DE_{out}
0	0	0	0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	1	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8
0	0	1	0	D_{23}	D_{22}	D_{21}	D_{20}	D_{19}	D_{18}	D_{17}	D_{16}
0	0	1	1	D_{31}	D_{30}	D_{29}	D_{28}	D_{27}	D_{26}	D_{25}	D_{24}
…	…										
1	1	1	1	D_{127}	D_{126}	D_{125}	D_{124}	D_{123}	D_{122}	D_{121}	D_{120}

Each of the eight outputs are completely determined by the combination of the four inputs

selected based on only 4 inputs (the D inputs); the C inputs have a special function, and are not involved in the selection of a truth table row.

Instead of being inputs into the truth table evaluation, the C inputs are used to control the writing and reading of a cell's truth table, as follows:

- If all of a cell's C inputs are 0 then the cell is said to be in D (or "data") mode. In this mode, the cell's outputs are derived from the truth table, using a row determined by the cell's 4 D inputs.
- If any of a cell's C inputs are 1 then the cell is said to be in C (or "control") mode. In this mode, the cell's outputs on each side depend on whether or not that side's C input is 1 or 0:
  - if the C input is 0 on a given side (called an "inactive side"), then that side's D output is 0, and its D input is ignored;
  - if the C input is 1 on a given side (called an "active side"), then that side's D output is determined by a bit of the cell's current truth table. Additionally, that side's D input is OR'd with the D inputs on all other active sides, and the resulting bit will replace the truth table bit that's being presented on the D output(s).

In the case of two adjacent cells ("SOURCE" and "TARGET," as shown in Fig. 29.6), SOURCE can interrogate TARGET's truth table as follows:

1. SOURCE asserts its own C output to its east, which asserts TARGET's C input from its west;

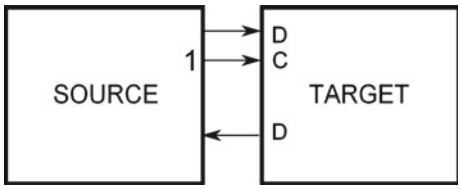

Fig. 29.6 An example of cell analysis. The cell on the *left* ("SOURCE") is examining the truth table of the cell on the *right* ("TARGET") by controlling TARGET's C input and monitoring TARGET's D output

2. TARGET will present a bit of its truth table to its D output on its west; and
3. SOURCE reads its D input from the east, which contains that truth table bit.

Cells map inputs to outputs asynchronously: that is, as soon as a cell's inputs change, its outputs will change according to the cell's truth table, without regard to any sort of synchronizing clock. In contrast, there is a system-wide clock used for C mode operations. Specifically:

- when a cell enters C mode, an internal (per cell) bit counter is reset to 0;
- the truth table bit indexed by this counter is presented to the appropriate D output(s);
- on the rising edge of the system clock, the appropriate D inputs are OR'd, and the resulting bit is saved internally;
- on the falling edge of the system clock, the saved bit it loaded into the truth table at the position indexed by the bit counter; the bit counter is incremented; and the new bit now indexed by the bit counter is sent to the appropriate D output(s).

This process continues as long as the cell remains in C mode.

In this way, the SOURCE cell in Fig. 29.6 can completely read the TARGET cell's truth table by simply asserting its own C output, and repeatedly reading its D input each time the system clock ticks low. Assuming it outputs 0's on its D output, the TARGET cell's truth table will be filled with all 0's after 128 clock ticks. Figure 29.7 shows a slightly different configuration by which the SOURCE cell can non-destructively read the TARGET cell's truth table. Figure 29.8 shows a further modification, in which a third cell ("DEST") is also placed in C mode, and configured to be a clone of the SOURCE cell.

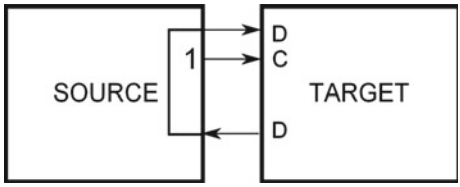

Fig. 29.7 Non-destructive read of a cell. The SOURCE cell re-circulates the truth table as it's read from the TARGET cell, so that TARGET's truth table is unchanged by the reading

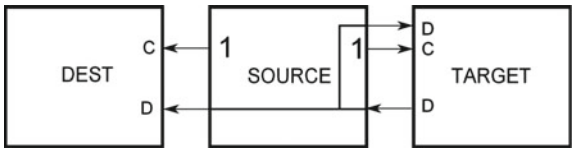

Fig. 29.8 Non-destructive cell replication. SOURCE reads TARGET's truth table, and simultaneously refreshes TARGET's truth table while also configuring DEST's truth table to be a copy of TARGET's. This effectively makes DEST a clone of TARGET

To create a Cell Matrix, cells are tiled in a 2-dimensional array, with each cell connected to four adjacent cells (one on each side), as shown in Fig. 29.9. This allows each cell to be potentially interrogated and/or modified by four neighbors; and also allows each cell to potentially interrogate and/or modify those same neighbors. It's important to note that *this is the full extent of the direct control any cell has over other cells within the matrix*. Of course, to be useful, cells must be able to interact with more than just a few immediately-adjacent cells. This takes place via intermediary cells. For example, one cell X may communicate directly with a neighboring cell Y, and may configure Y to allow X to control one of Y's neighbors Z (Fig. 29.10). This type of multi-cell control is typical of circuits built on the Cell Matrix. Several specific examples are presented in the next section.

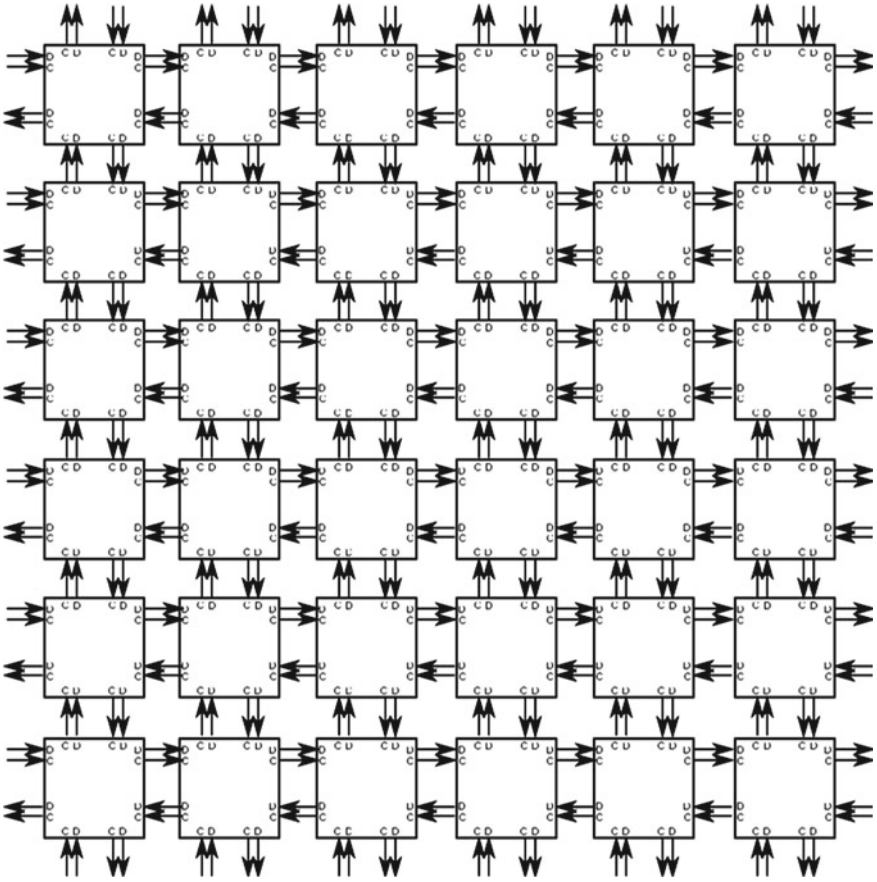

Fig. 29.9 A two-dimensional 6 × 6 Cell Matrix. Each cell communicates only with its immediate neighbors. Edge cells are accessible from outside the matrix

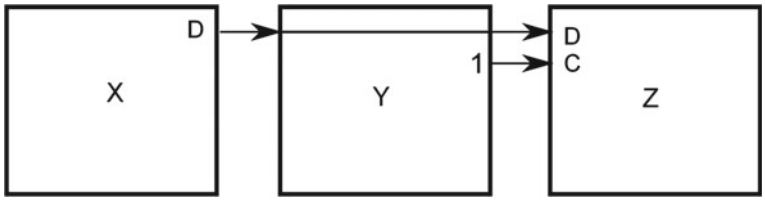

Fig. 29.10 Non-adjacent cell configuration. Cell X is using its neighbor (Y) to configure non-adjacent cell Z

The limitation of nearest-neighbor communication is a key benefit to the architecture, since it allows the system to scale nicely. Connecting two matrices along their edges results in a larger matrix. This has important implications for manufacturing large Cell Matrices, especially with techniques that employ self-assembly [15, 16].

While a 2D matrix is more-easily explained and analyzed than a 3D one, the architecture itself is agnostic to dimensionality and interconnection topology. One can also implement, for example, a 2D matrix with 3-sided cells, or a 3D matrix with 4-sided cells. The most common 3D implementation is using 6-sided cells (cubes) [11], though research is ongoing using 8- and 14-sided cells [18]. Higher dimensionalities are also possible, though scalability of the matrix is generally impaired at higher dimensions, as the matrix generally can't be extended without modifying the interconnection network in already-assembled sub-matrices.

29.3 Approach

In this section, several examples are presented of Cell Matrix circuitry that can be used for introspection and modification of digital circuits.

29.3.1 Multi-channel Wires

The first example of a modification circuit is a *multi-channel wire* [10, 14]. This is a circuit that can be used to establish and exercise control over a remote area of cells. Figure 29.11 shows a wire with three channels:

- The Program Channel ("PC"), which is used to control a target cell ("*")'s D input;
- the Control Channel ("CC"), which is used to control a target cell's C input; and
- the Break Channel ("BRK"), which is used to carry additional C or D information.

Each channel consists of a number of single cells. Figure 29.12 shows the details of the cells comprising the PC and CC channels. As can be seen, sending a 1 or 0 down PC causes the same value to be sent to the target cell's D input. The PC is

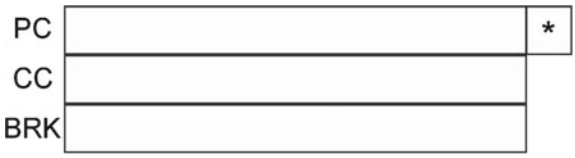

Fig. 29.11 A three-channel wire, used for controlled access to a remote cell ("*"). Each channel is a $1 \times n$ collection of n single cells. The PC is used to control *'s D input, and CC is used to control *'s C input. BRK is an auxiliary line, generally used to break or retract the wire

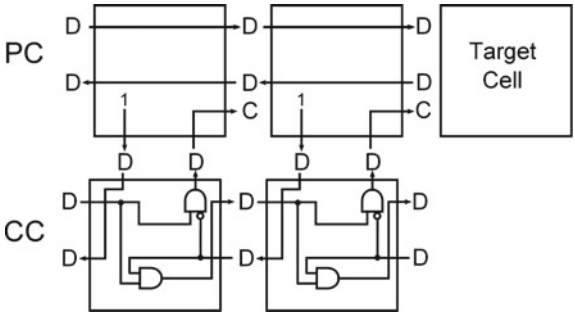

Fig. 29.12 Details of the cells comprising the PC and CC. A wire of length 2 is shown. The PC always transmits data to the D input of the Target Cell. The CC transmits a signal from cell to cell; at the end of the channel, the CC also transmits a 1 to the PC head cell's southern D input, which the PC head cell transmits to the target cell's C input

usually bi-directional, so that the target cell's D output is delivered through the PC to its own western output.

Similarly, sending a 1 or 0 down the CC drives the target cell's C input. The operation of this channel is more complicated than that of the PC. The CC employs a feedback signal to determine where the head of the wire is. Each PC/CC pair works together to route a signal to the previous pair's CC cell. Any CC cell receiving this signal simply routes its western D input to its eastern D output; but in the absence of this signal, the CC cell routes its western D input to its *northern* D output, which the PC then uses to drive the target cell's western C input. The final effect of this mechanism is that in a line of PC/CC cells (running west-to-east), the easternmost pair acts differently from the other cells, working together to control the target cell's mode. It's assumed here that the cell to the south of the target cell is empty (or at least is sending a 0 to its western D output). If that is not the case, then this cell must be *pre-cleared* before extending the wire.

By building a chain of PC and CC cells, placed side-by-side as in Fig. 29.12 (which shows a wire of length 2), one set of cells can thus control a non-adjacent cell's C and D inputs and read its D output. This allows circuity in one region of the matrix to interrogate and modify remote cells, provided a multi-channel wire exists between the source and the target cell. This leaves the question: how does one construct a multi-channel wire?

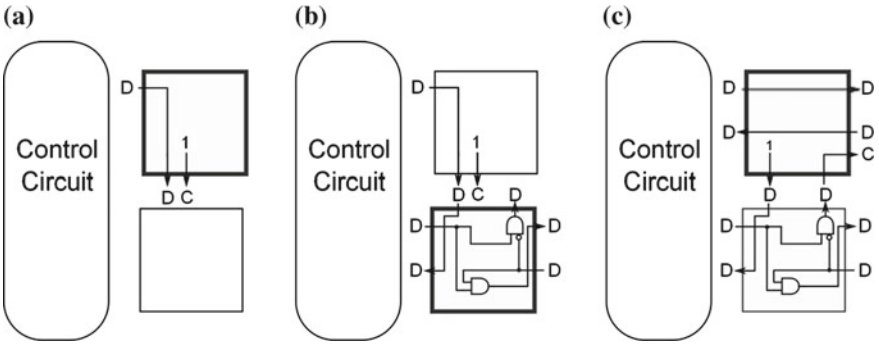

Fig. 29.13 Construction of an initial PC/CC pair. In each sub-figure, the cell being configured is indicated in bold. First **a** the control circuit configures the northern cell so that it can be used in **b** to configure the southern cell as a CC cell. In **c** the northern cell is reconfigured, this time as a PC cell. After these steps, a wire of length 1 has been built

The answer is that **a multi-channel wire can be used to construct a multi-channel wire**. By beginning with access to a single pair of cells, one can build an initial PC/CC pair in three steps, as shown in Fig. 29.13:

1. in (a), the top cell is configured so as to allow configuration of the lower cell;
2. in (b), the lower cell is configured as a CC; and
3. in (c), the top cell is configured as a PC.

Note that only access to the northernmost cell's C and D inputs is required. Since that cell is used to configure the cell to its south, no direct access is required to that southern cell.

This is called a "wire-building sequence," or simply a "sequence." A sequence is generally any collection of configuration operations, typically where certain cells are alternately configured and then used to configure other cells. Later, sequences of sequences ("super-sequences") will be used for more complex configuration operations, such as configuring 2D or 3D regions of cells.

Given an initial wire of length 1, the above steps can be repeated, since the control circuit *now has access to the non-adjacent target cell* (Fig. 29.14). Once access is available to that new target cell, it can of course be used to configure yet another PC/CC pair, and so on. In this way, wires of arbitrary length can be built, thus allowing access to remote regions of the matrix. Note that the key to performing this wire extension is the feedback path in the CC cell. Following the third step in the above sequence, the PC sends a 1 into the CC, which then feeds it back to the previous stage's CC, which thus ceases acting as the final cell ("head cell") in the wire, and simply routes its western D input to its eastern D output. This is the basic mechanism by which wire extension is possible.

Figure 29.15 shows a circuit for turning a wire. In this case, a wire that was extending from west to east is now turned to the south. This 2×2 circuit works effectively the same as the simple PC/CC pair, but the feedback path is modified

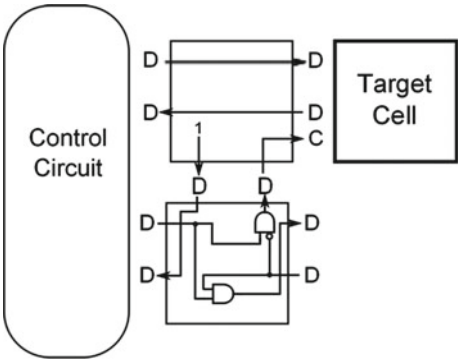

Fig. 29.14 Once a wire of length 1 has been constructed, the Control Circuit can use it to configure a non-adjacent Target Cell. By repeating the steps shown in Fig. 29.13, the Control Circuit can extend the wire to a length of 2. This extension can be repeated as many times as needed to make wires of arbitrary length

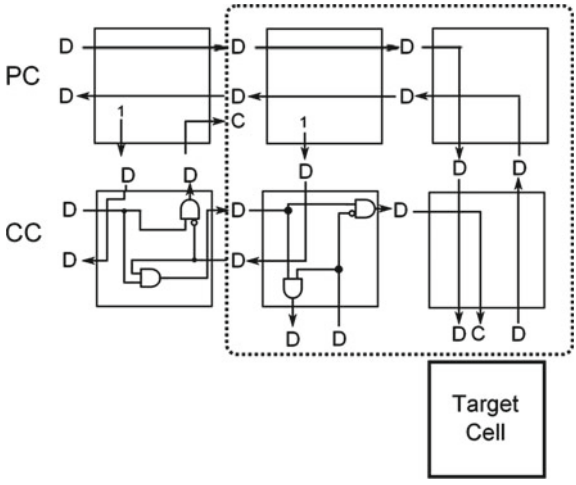

Fig. 29.15 Circuit for turning a 2-channel wire. The PC and CC control the Target Cell. The wire can now continue to extend to the south

slightly, so as to allow the feedback 1 signal to be presented following the final configuration step. Constructing a corner is a 12-step process: details can be found in [10, 14].

The use of a BRK line is sometimes useful for interrupting a wire, as shown in Fig. 29.16. In this example, a wire of length 4 has been constructed, granting access to the target cell marked "*". However, two cells of the BRK line have also been constructed adjacent to the 2-channel wire. The leftmost cell simply routes D from west to east; but the second cell of the BRK line routes its western D input to its northern C output. Thus, if a 1 is sent down the BRK line, it will clear the second cell of the control channel. The effect of this is to interrupt the feedback from the second PC cell, thus causing the first CC cell to become the new head CC cell (which in turn causes the first PC cell to become a head cell as well). This effectively causes the cell marked "+" to become the new target cell of the wire. This mechanism can thus be used to restore control to a prior location after a wire has been extended. This may be used, for example, in a bootstrap super-sequence [11].

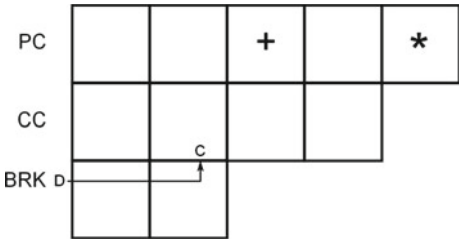

Fig. 29.16 Example of a break line. The PC/CC pair initially give access to the target cell ("*"). By asserting the BRK line, the second CC cell can be cleared, changing the location of the target cell to the cell marked "+"

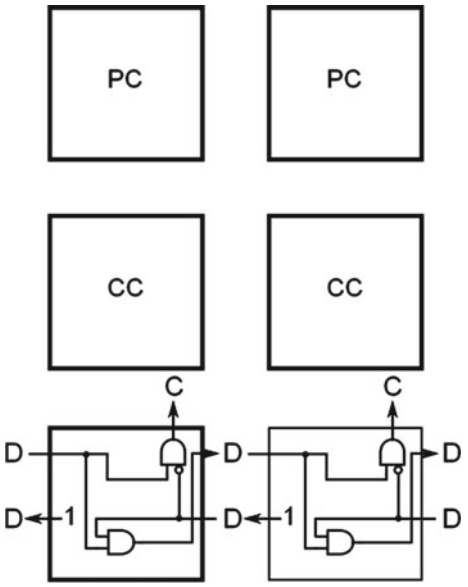

Fig. 29.17 A breakable wire. The BRK channel is dynamic: whatever its length, the rightmost cell will route its western D input to its northern C output, thus clearing the CC cell above it. This changes the wire's target cell to be the cell directly above the end of the BRK line. The target cell can then be configured to clear the last cell on the BRK line, thus retracting it

In some cases, rather than simply breaking a wire, it may be desirable to back a wire up one step. Figure 29.17 shows a *retractable wire*: a circuit to allow a wire to be retracted, one cell at a time. This is useful for synthesizing a circuit in reverse: extending a wire, configuring the target cell, then backing up one step and configuring the new target left in the wire's wake, and so on.

The key to a retractable wire is the dynamic BRK line. Instead of a line of cells that simply route data from one side to the other, each cell of this BRK line actively determines whether it is the last cell in the channel: if not, it simply routes data from west to east; but if it sits at the end of the wire—i.e., if there is no BRK cell to its right—then it routes data from the west into the C output to the north.

The wire is extended in the usual way: the target cell (in front of the easternmost PC cell) is configured to build a new BRK cell, then a new CC cell, and finally a new PC cell. This allows the wire to be extended as far as desired. To back the wire up,

the BRK signal is activated to clear the head CC cell, which moves the target cell back one spot. The new target is then configured to clear the current head BRK cell. This allows the target to be configured, but leaves the BRK line ready to clear the previous stage of the wire, thus allowing further retraction of the wire.

29.3.2 Multi-headed Wires

The above circuits are used to configure a single target cell at a time. However, modified versions allow parallel configuration of multiple target cells, as shown in Fig. 29.18. In this figure the PC and CC inputs drive the D and C inputs, respectively, on a series of target cells (marked "*"). Such a wire (with multiple heads) is called a "Medusa Wire," and allows multiple targets to be configured simultaneously. Of course, since there is only a single PC/CC pair delivering information to each target cell, all target cells will be configured identically. However, *identical circuitry built near each target cell can be used to effect non-identical synthesis* by using techniques such as local indexing (discussed at the end of this section).

A Medusa wire is not easily extended in the same manner as a single-headed multi-channel wire. However, a multi-channel wire *below* the Medusa wire can be used to synthesize the Medusa. And once built, the Medusa wire allows configuration of a line of identical circuits. Since the Medusa wire itself is a line of identical circuits, it follows that *the Medusa wire can be used to build another Medusa wire* (directly below the original). Similarly, that Medusa can be used to build a 3rd copy, and so on (as shown in Fig. 29.19). A column of additional control circuitry is also built

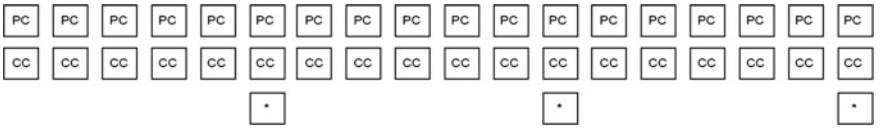

Fig. 29.18 A multi-headed Medusa wire. The PC and CC control access to the D and C inputs of multiple target cells (each labeled "*")

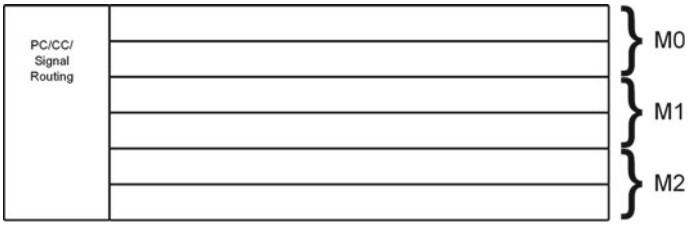

Fig. 29.19 Medusa wires built from Medusa wires. In this example, $M0$ was built first (sequentially), and then used to build $M1$ *in parallel*. $M1$ was then used to build $M2$ in parallel. This process can be repeated for more-efficient configuration of a region of cells

along the left edge of the assembly, to route the PC/CC signals to all Medusa wires. While the first wire—assuming it contains n heads—takes $O(n)$ steps to build, the second wire takes $O(1)$, as does the third, fourth, and so on. This allows n^2 heads to be configured in $O(n)$ steps.

By themselves, these heads have limited purpose if they're only used to build more Medusa wires. In practice, this 2D plane of Medusa heads might be used to configure a 2D plane of target cells above the Medusa plane (up the Z axis in a 3D Cell Matrix, for example). This utilizes a modified Medusa head circuit, which can be switched between multiple operating modes.

Alternatively, a scheme such as shown in Fig. 29.20 can be used. In (a), a single-headed multi-channel wire has been built, and as it was extended, it was used to synthesize the pieces of a Medusa wire ("MED0"). MED0 is a multi-headed wire that configures regions to its north or south, depending on the value of a direction signal. In (b), MED0 is used to synthesize a 1D collection of 2D target circuits to its

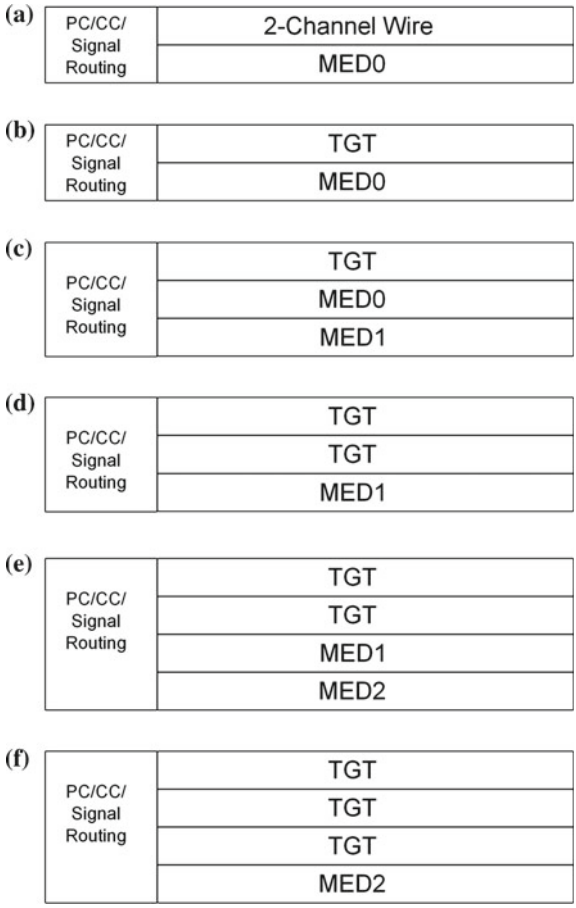

Fig. 29.20 Parallel build of target circuits and new Medusa wires using Medusa wires: **a** a standard two-channel wire has been used to build Medusa wire MED0; **b** MED0 configures the region to the north into a row of desired target circuits; **c** MED0 configures the region to the south into a new Medusa wire MED1; **d** MED1 configured a second target region; **e** a third Medusa (MED2) is configured; and **f** MED2 configures a third target region

north (for example, pieces of a finite-element analysis system [19]). In (c) MED0 is directed south to configure a new Medusa wire ("MED1") to its south. In (d), MED1 configures regions to its north, creating another row of target circuits. In (e), MED1 configures a new wire "MED2" to its south, and in (f) MED2 configures a third row of target circuits to its north. This process repeats, configuring (in parallel) new rows of target cells, leading again to an efficient initialization of n^2 regions of target cells in $O(n)$ time steps. For simplicity, the target regions are shown as being thin, but in practice taller regions can be configured by placing the Medusa wires further to the south.

29.3.3 Analysis of Cells: Intrinsic Operations

The above techniques and circuits are useful for synthesis or modification of circuits within the Cell Matrix. Another useful task is the *analysis* of circuits or states within the Cell Matrix. The simplest example of this is responding to outputs from neighboring cells. Since each cell's outputs are directly connected to the inputs of any neighboring cells, each cell automatically responds to its neighbors outputs according to its own truth table configuration.

A more interesting example is a cell that reads another cell's truth table, by asserting the target cell's C input and reading that cell's D output. This is also a basic cell-level operation within the matrix. Once read, a cell's truth table can be stored (perhaps in another cell's truth table memory, or in a larger structure composed of many cells acting in concert as, say, a shift register). Truth tables that have been read can be compared to templates (useful for reverse-engineering dynamically-built circuits); analyzed for patterns (to examine the results of, say, random generation of configuration strings); rotated (to build a new circuit that faces a different direction); combined with other truth tables (to breed two configuration strings in a genetic algorithm [17]); and so on.

29.3.4 Circuits and Techniques for Testing Single Cells

Moving away from intrinsic cell-level operations, one can create test circuits and test patterns. Figure 29.21 shows a simple arrangement whereby one cell—called the source ("SRC")—may test a target cell ("*"). A basic test is to load the truth table for the equation $DataWestOut \leftarrow WestIn$ (abbreviated $DW = W$) into the target cell, and then examine the target's D output as the SRC sends 0's and 1's into the target. The expected behavior is that the target echoes back whatever value the SRC sends. This can detect errors where the target's DW_{out} is stuck-at-1 or stuck-at-0 (or where there's a problem configuring the target at all).

Next, the target cell be configured with the equation $DW = !W$ (e.g. an inverter), and the above test repeated. This test can detect shorts between the target's DW_{in} and

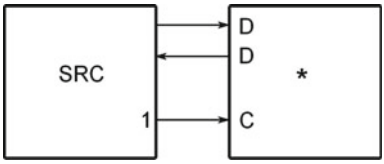

Fig. 29.21 Simple cell-testing arrangement. "SRC" is able to load test patterns into "*," send values to *'s D input, and monitor its response on *'s D output

DW_{out}. Additional tests can be used to check other aspects of the target's D mode behavior (though each of these tests is also to some degree checking the target's configurability).

C mode can be more-thoroughly tested by loading different bit patterns into the target's truth table memory, and immediately reading back those patterns (without letting the target leave C mode). If the test pattern reads correctly, then the SRC knows that the target's memory can successfully store the given test pattern (and can also successfully enter C mode, index bits in the truth table, and so on). By using different patterns (all 0's; all 1's; alternating 1's and 0's; alternating groups of 1's and 0's, arranged so that each physical row of the truth table's memory stores complementary values; and so on), different possible failure modes can be tested.

As an example, consider Fig. 29.22 which shows a potential layout for the 128 bits of a truth table inside a cell. Here, the memory is physically arranged as a 16×8 array of single-bit memories. For this example, assume there is a defect involving the 6th cell on the 3rd and 4th rows, whereby their outputs are shorted together. This defect will cause those cells to report the same value as each other. If we attempt to load the memory with the pattern

11111111000000001111111**1**1100000**0**0011111111000000001111111100000000

the boldfaced bits will be read-back as the same value as each other, which will indicate to the testing circuit that there is a problem in the memory storage subsystem.

29.3.5 Testing Regions of Cells

After exercising a cell's capabilities *sufficiently-enough* to determine that the cell is usable, that cell can then be used to test additional cells. This is a process similar to bootstrapping a 2D region of cells: first one cell is tested; then a second cell; then these cells are configured as a wire, and used to test a 3rd and 4th cell, which are used to extend the wire, and so on. As more cells are tested (and found to be good), more control can be exercised over the region being tested [4].

In any defect-testing system, there is always a concern that the test circuitry itself may be defective. In the case of an introspective, self-modifying system such as the Cell Matrix, these concerns are largely obviated, since the test circuitry is not a

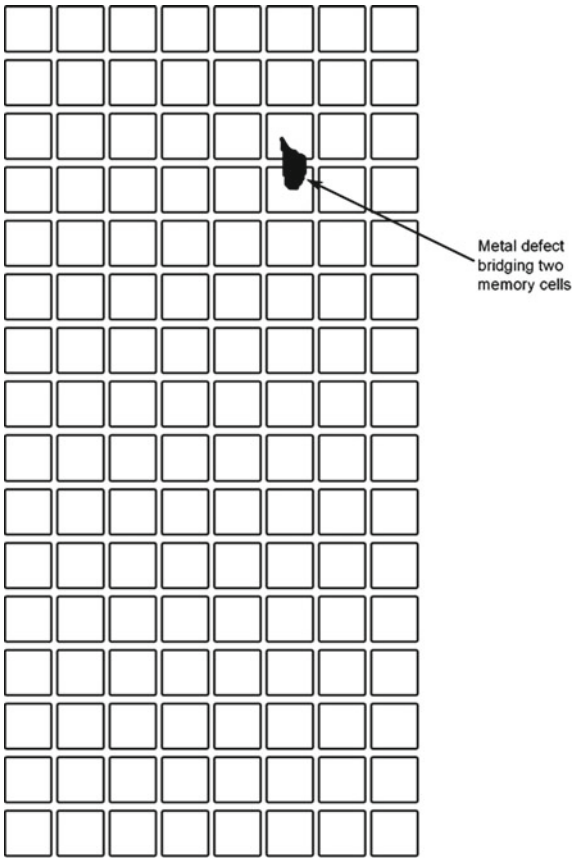

Fig. 29.22 Example of memory defects inside a 16 × 8 truth table. Here, there is a short between the 3rd and 4th rows, in the 6th cell on each row. This can't be detected by loading all 0's or all 1's, but would show up when loading a repeating pattern of 8 0's followed by 8 1's

pre-existing, hardwired system, but rather is constructed, cell by cell, *as each cell is subjected to fault testing and found to be working correctly*. Thus, if a cell is found to be defective, it is simply not used in constructing the test circuit (of course, there are extreme edge cases, such as if every cell along the outer perimeter of the matrix is defective, in which case there is no way to reach any internal cells). If a wire is being built, and encounters a defective cell, the wire can be backed-up two steps, and a corner built to move down past the defective cell, and so on.

Testing can also be done from different sides, as shown in Fig. 29.23. Since wires can be bent and turned, a cell being tested can be approached from different sides, to allow testing of the I/O lines on all sides of a cell.

Note that it is generally not feasible to test a system for *every* possible failure mode. For example, there are 2^{128} possible ways to arrange a 4-sided cell's memory, and numerous possible present states inside a cell's circuitry. While these are too numerous for exhaustive testing, it may be possible to test many of the states in which a cell is expected to actually operate. For example, due to the layout of cells

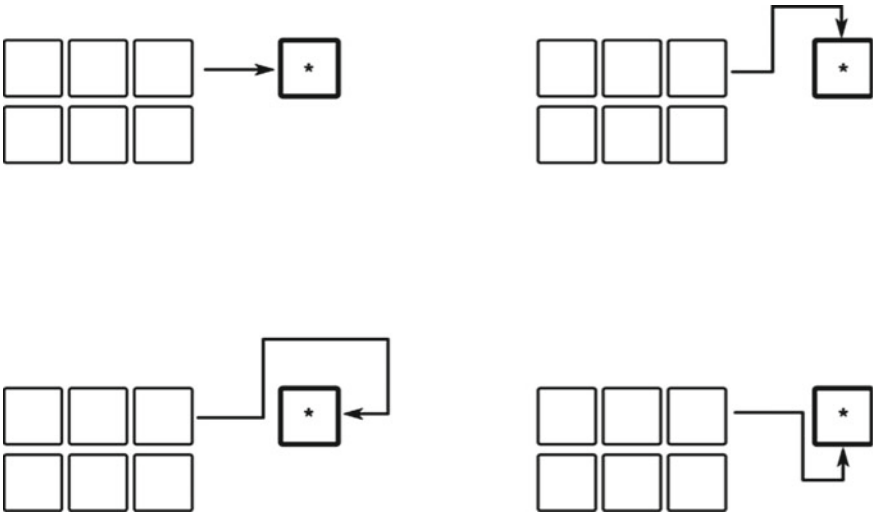

Fig. 29.23 Interrogating a cell from different sides. By making a series of turns, a multi-channel wire can access a target cell ("*") from different sides. This provides a greater testing capability, since I/O lines on each side can be examined

within a circuit, it may not be possible to interrogate the cell from (say) its eastern edge. In that case however, it's likely that the circuit that's being built may not actually use the cell's eastern edge, in which case an I/O error associated with that side may not be relevant to the expected operation of the circuit being constructed.

29.3.6 Isolating Defects

The above techniques are primarily concerned with **fault detection**: identifying when one or more cells are defective and may potentially impair the behavior of a circuit built with those cells. An additional consideration arises given the possibility that a defective region of cells could inadvertently affect nearby cells in an undesirable way. Since cells can configure other cells, and cause those cells to configure still other cells, it's theoretically possible (though unlikely under normal circumstances) that a defective cell could alter the configuration of nearby (or even remote) non-defective cells. To prevent such a situation, one can construct circuits (*from non-defective cells*) that isolate defect-free regions of cells from regions containing defects. One such circuit is called a "Guard Cell." A guard cell employs a second cell, as shown in Fig. 29.24. The middle cell is the guard cell, but it is controlled by the cell on its left, which is simply placing the guard cell into C mode. Under the Cell Matrix architecture, when a cell is in C mode, it is unable to assert any of its C outputs. Since the guard cell is built from a non-defective cell, the guard cell is guaranteed to

Fig. 29.24 Example of a *guard cell*. The two cells on the *left* work together to prevent the *rightmost* cell ("?") from being able to affect the cells on its *left*. Since the *middle* cell is always in C mode, it can never assert its own C outputs, and thus can't be used by cell "?" to configure other cells

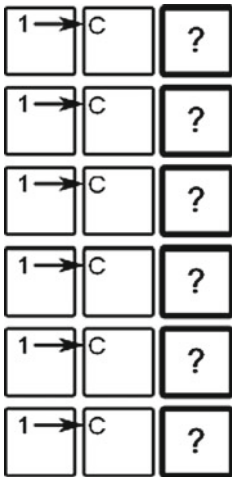

Fig. 29.25 A line of guard cells acts collectively as a *guard wall*, which protects one region of the matrix (to the *left* of the wall shown here) from the behavior of another region (immediately to the *right* of the wall shown here)

be outputting $C = 0$ on all sides. Thus it is impossible for the defective cell (marked "?") to use the guard cell to configure other cells.

A line of guard cells constitutes a "Guard Wall" (Fig. 29.25); and a closed perimeter of guard cells (Fig. 29.26) creates a "Guard Ring" that isolates a region of cells from the rest of the matrix.

29.3.7 Auto-Location Within the Matrix

Another use of introspection is in determining ones location within a collection of identical cells. Since parallel-configuration techniques such as Medusa wires lead to the creation of a number of identical sub-circuits, these circuits don't intrinsically have any identifying information that can be used to self-identify as being distinct from other circuits. Such self-identification is an important first step in subsequent differentiation of the matrix into heterogeneous components.

Figure 29.27 shows a line of identical circuits, each containing an increment unit. Each circuit receives—either in parallel or as a serial stream—an integer from the circuit on the left; adds one to it; and sends the incremented value to the circuit on its right. Assuming the leftmost circuit is receiving all 0's from its left (either because it is adjacent to a region of unconfigured cells, or because it is at the edge of the

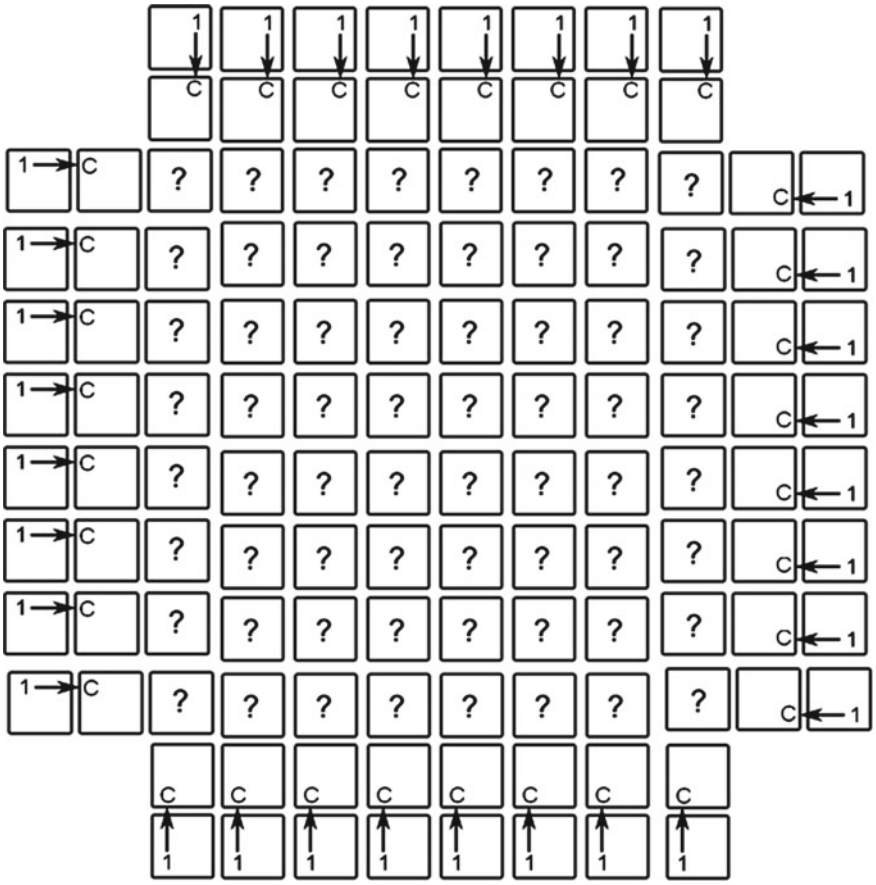

Fig. 29.26 A closed ring of guard cells creates a *guard ring*, which isolates the behavior of cells (labeled "?" here) inside the ring from the rest of the matrix

matrix), that circuit's X value will be 0, while the circuit to its right will see $X = 1$; the next circuit to the right will see $X = 2$; and so on. This generated value of X effectively tells each circuit what column it sits in. Similar circuitry can be used in the Y direction to allow cells to determine which row they occupy in a 2D collection of identical circuits. Given such positional information, circuits can then differentiate into pieces of a larger target circuit according to the shared circuit map. The circuits can also dynamically determine an optimal configuration for a target circuit based on available resources [13].

The above scheme can be simplified to let cells determine whether or not they are adjacent to an edge of the matrix. Circuits can read from (as in Fig. 29.28 for example) the north, and output a signal (1) to the south. If placed top-to-bottom, each circuit will read a 1 from the north, except the circuit placed at the very top of the matrix. Repeating this in different orientations, circuits can in this way identify

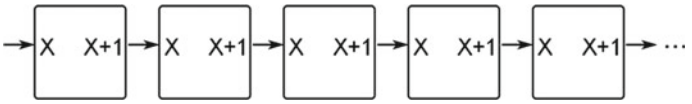

Fig. 29.27 A line of identical circuits that can locate themselves within the matrix. Each circuit (which is a collection of cells operating as a single unit) receives an integer from the *left*, increments it, and outputs the incremented value to the *right*. The leftmost circuit will be the only one receiving a 0 (assuming it's either at the edge of the matrix, or is adjacent to a region of cells that is not participating in this behavior). Every circuit will receive a unique value of X from its neighbor (provided enough bits are used in transmitting the integer). This allows circuits to identify themselves with a unique index relative to all other circuits in the collection

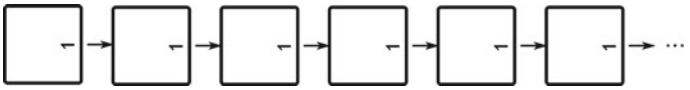

Fig. 29.28 Detecting the northern edge of the matrix. Even though each cell is configured identically, only one cell (the northernmost one) will receive a 0 on its northern input. This allows cells to identify when they are located at the top edge of the matrix

the boundaries of the matrix. A different scheme is to simply build a multi-channel wire, constructing a simple feedback test circuit in front of the wire as it extends (Fig. 29.29). While this arrangement is normally used for fault detection, it will also indicate where the end of the matrix is, since attempts to configure a feedback cell ("F") will fail beyond the edge of the matrix. By repeating this analysis on multiple rows, one may determine the location (and shape) of the matrix's edge. Note that this edge may be the physical edge of the matrix, but if a region of cells are defective, this will define the *effective* edge of the matrix.

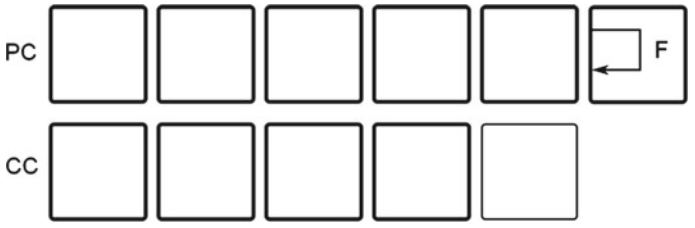

Fig. 29.29 Another example of edge detection in the matrix. A two-channel wire is grown to the right. At each step, a feedback cell ("F") is configured; a test ping is sent; and an echo is detected. The echo detection will fail when the wire has reached the rightmost edge of the matrix (since there is no cell available for building the feedback cell). This allows a circuit to determine the horizontal width of the matrix. A similar technique can be used to determine the matrix's height

29.4 Applications

The above sections have covered some basic underlying mechanisms that support self-analysis and self-modification. This section presents examples of digital circuits that utilize introspection to drive self-modification. These specific applications themselves are not necessarily "killer apps" for this technology. The goal is to present a set of examples illustrating ways in which this technology can be used to achieve unique, potentially interesting behaviors.

Four examples are presented:

1. An overflow-proof counter. This illustrates the use of self-modification to handle specific, expected conditions.
2. An autonomous circuit-scrubbing system. This system employs extensive self-modification, but also utilizes self-analysis of circuit configurations, to develop a system that can **repair** soft errors.
3. A system for parallel synthesis of circuits, including detection and avoidance of defective regions. This illustrates the use of introspection and self-modification to **work around** hard errors.
4. A system for detection of mis-oriented cells in a Cell Matrix: a system that **adapts-to and works-with hard errors**.

29.4.1 An Overflow-Proof Counter

Figure 29.30 shows a simple ripple-carry counter based on negative edge-triggered toggle flip flops. The counter is driven by the clock input on the right, and as each stage flips from 1 to 0, it toggles the next stage to its left. The configuration shown is a 3-bit counter, which counts from 000 to 111 before overflowing and returning to 000.

Figure 29.31 shows the same circuit extended by adding a fourth flip flop to the left. Adding this additional stage changes the circuit to a 4-bit counter. Such an extension requires no additional changes to the pre-existing circuit; the change is made only on the leftmost edge, by feeding the current counter's MSB into the new stage's clock input.

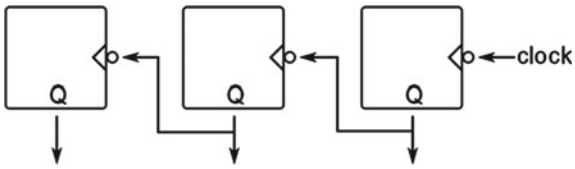

Fig. 29.30 Basic 3-bit ripple-carry counter. Each T flip flop toggles when the prior bit changes from 1 to 0. After 8 ticks, this counter overflows from 111 to 000

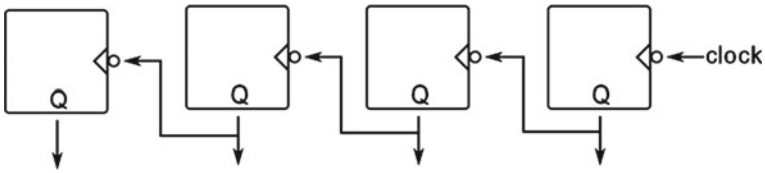

Fig. 29.31 By simply adding a 4th flip flop to the *left* of the MSB, the counter is now extended to a 4-bit counter, and can count to 1111 before overflowing to 0000

When implemented on a Cell Matrix, a toggle flip flop requires only three cells, using the configuration show in Fig. 29.32. The middle cell includes a pair of 1-bit multiplexers which select either their 0 or 1 input (based on the value of the *S* input).

The top and bottom cells provide a feedback path, while the middle cell receives the incoming clock and alternates the bit that's circulated among the cells. The bottom cell delivers the flip flop's Q output, which is also sent from the middle cell's left edge. This allows a new stage to be added by simply placing it adjacent to the prior rightmost stage (assuming those cells aren't already being used for some other purpose).

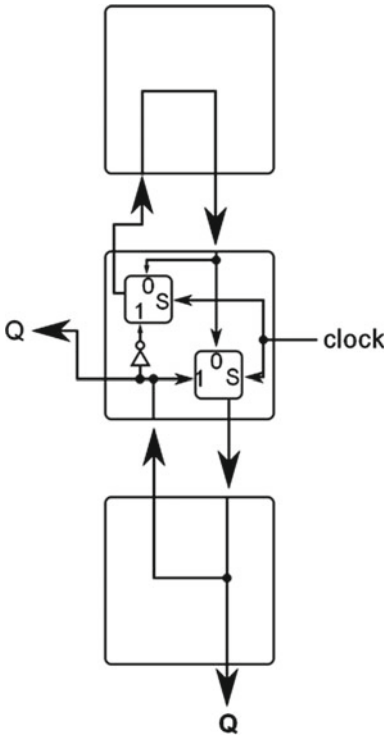

Fig. 29.32 3-cell implementation of a T flip flop. The clock is applied to the middle cell's eastern D input. The flip flop's output (Q) is sent to the bottom cell's southern D output. It's also copied to the middle cell's western D output, where it can be applied to the clock of another T flip flop positioned directly to its left. The "S" blocks in the middle cell are one-bit multiplexers

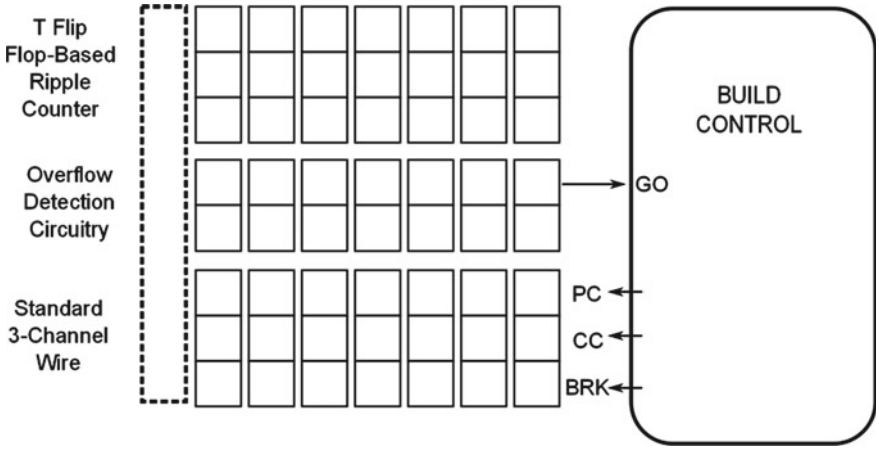

Fig. 29.33 An expanding counter. In addition to an *n*-stage ripple counter (comprised of *n* T flip flops), there is a set of circuitry located below each flip flop. This circuitry determines if the two MSBs of the count are 11, and reports that fact to the main control circuit (via Build Control's "GO" input); it also routes build commands from Build Control to the edge of the flip flop collection, allowing Build Control to construct a new flip flop (in the *dashed* region). This allows the counter to grow in size dynamically, as necessary to prevent overflow

Figure 29.33 shows a toggle flip flop-based counter with additional hardware to allow it to automatically build more stages as the count increases. Below each flip flop is an *overflow-detection* circuit with three functions:

1. it contains an edge detection circuit to determine if it is located below either of the two leftmost flip flops;
2. it contains simple logic to determine—if it is located below either of the two leftmost flip flops—whether each of those flip flops is currently outputting 1; and
3. it contains a piece of a two-channel wire which can be used to configure cells to the left of the leftmost flip flop.

Figure 29.34 shows the details of this overflow-detection circuit.

The logic in step 2 above is fed through the collection of circuits to the state machine located to the right of the flip flop assembly. When the *GO* input is triggered (raised from 0 to 1), the state machine begins generating supersequences. Those are sent down the two-channel wire (function 3 described above), and cause the synthesis of a new flip flop and overflow-detection circuit to the left of the current assembly. Once the flip flop has been built, the wire itself is reconfigured to extend it another stage and re-build a new corner at the leftmost edge, thus preparing for the next future build operation.

Since the build is triggered when the leftmost two bits are both 1, it happens when the counter has reached approximately 75 % of its maximum value. As long as the build can be completed in less time than it takes the counter to count the remaining 25 % of its maximum value, the build will finish before the counter overflows, i.e., a

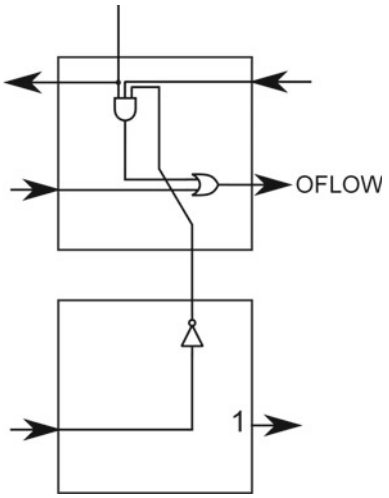

Fig. 29.34 Overflow-detection circuitry. The lower cell determines if it is at the leftmost-edge of the collection, and outputs a 1 to the north if so. The upper cell uses this 1 signal, along with the counter bit from above and from the right, to determine if an overflow is going to occur soon. The resulting "OFLOW" signal is routed from west to east through these same cells (via the OR gate). This upper cell also routes the current stage's data to the western cell

new flip flop will be present before it needs to operate in order to maintain the correct count. This circuit thus implements an *overflow-proof* counter.

Of course there are some details to be noted here. The first is that this is a ripple counter, and may not be suitable for counting to extremely large values. The design, however, can be easily modified to create a synchronous output. Likewise, preset and clear inputs can be incorporated, as well as other embellishments; the basic design of the expanding element remains essentially the same.

This scheme assumes there are available cells on the left of the counter. One may ask, if there are free cells there, why not just pre-configure them to be counters, rather than waiting until they are needed? The answer is that one may not know ahead of time how large a counter is needed. Until those extra cells on the left are used for the counter, they are available for use by other circuits. This raises another question: what happens if the circuit tries to expand, but some other circuit is using the needed cells? The answer is that something must be relocated: either the circuit currently using the needed cells, or the counter itself. This may seem a daunting managerial task, but it is in many ways no different from how a typical memory management system works in a modern operating system. This raises the possibility of implementing a *hardware management system*, perhaps incorporating constructs similar to virtual memory, paging, swapping, and so on.

This technique thus makes good use of self-modification to respond to detected conditions in the environment (in this case, the approach of an overflow condition). In this case, "introspection" is limited to analysis of the system's state. The next example introduces a circuit that uses self-analysis of its own memory configuration.

29.4.2 Triple Redundancy with Autonomous Defect Scrubbing[1]

A simple defect detection and correction technique is to use *triple redundancy* [9], where three independent copies of a digital circuit operate simultaneously. Assuming defects are relatively rare, i.e., if there is a high likelihood that only one circuit will fail at a time, one can compare the states of the three copies, and in the case of a discrepancy, the majority state is taken to be correct. While relatively simple to implement, a triple-redundant system is only useful for detecting a single defect. Once one of the three copies is corrupt, the system now has only two pristine copies, which does not allow determination of a majority state if one of those remaining circuits becomes corrupt.

Some errors are transient—for example, a momentary voltage spike in the output of a logic gate—and will disappear shortly after they appear. Their *effect*, however, may be longer-lasting, particularly if they occur in part of a memory circuit. If they occur in the configuration memory of a reconfigurable device, their effect may become permanent (until the system is reconfigured). While reconfigurable systems are potentially well-suited to using triple redundancy (since, by employing a sufficiently-large device, there may be enough space to make three copies of the target circuit, and to incorporate the necessary mechanisms for voting), they suffer from this particular vulnerability in the event of a *configuration memory upset* [2].

On a substrate that allows analysis of configured circuitry, one can implement a circuit that does more than simply compare three copies of a circuit. One can actually compare three copies *of the circuit's configuration memory*, and, upon determining that there is a bit error somewhere, *correct that error* to restore the circuit to pristine condition.

Figure 29.35 shows a system that uses these concepts to mitigate memory upset-related faults. $C1$ $C2$ and $C3$ are three copies of the target circuit. Each block labeled "Wire Head" contains the circuitry for a small multi-channel wire, which can be used to build longer multi-channel wires. The control circuit ("CTRL") directs these wire-building operations, extending these wires across the top of each circuit ($C1$–$C3$). As the wires extend, the configuration memory of the cells along the top of each circuit is read and conveyed to the control circuit. The control circuit performs a bit-by-bit comparison of the three configuration strings, checking to see whether all three strings agree. In the case of a discrepancy, the majority bit value is taken to be correct. As the wires extend across the circuits, the control circuit stores these (corrected) configuration strings in the block labeled "Storage," until the wires reach the rightmost edge of the circuits. At this point, the storage circuit contains a (presumably) pristine copy of the configuration memory for the top row of the circuits. Next the wires are de-constructed (with the end of the wire moving from the

[1]This work was supported by DOE/LANL under subcontract 90843-001-04 with the Regents of the University of California.

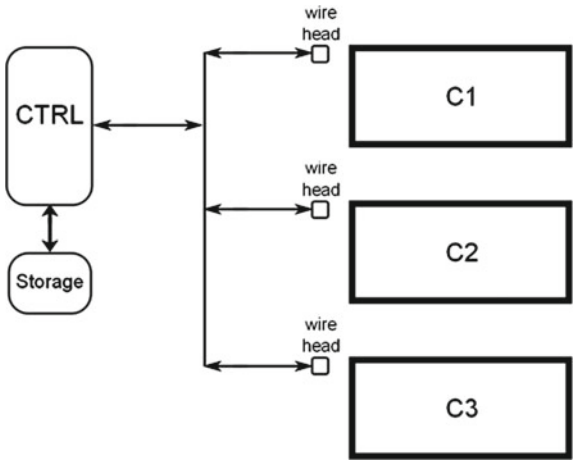

Fig. 29.35 Overview of triple-redundancy system. The CTRL circuit guides the construction of three wires which scan three copies ($C1$–$C3$) of the target circuit. By comparing the truth tables in each copy's cells, temporary upsets in the truth table memories can be detected and corrected by the CTRL circuit

rightmost edge back to the initial position at the Wire Head), and the stored configuration information is used to re-build perfect circuitry in its wake.

Following the above operations, the controller builds wire from each wire head to the south, turns counter-clockwise 90 degrees, and begins to move across the next rows of circuits $C1$–$C3$, again reading, voting, saving, and finally re-building their configuration memories. This process is repeated, until the entire circuits have been "scrubbed" using this technique. In the end, any upsets in the configuration memory of a circuit should have been corrected.

Figures 29.36a–d illustrate the first row of this build process. The process illustrated in these figures is repeated in all three circuits $C1$–$C3$ (from Fig. 29.35). In Fig. 29.36a, the circuit is intact, and the three cells marked "*" are about to be examined via a set of three three-channel wires. In Fig. 29.36b three cells in the upper-left of the circuit have been read, and are now a part of the extended wire. In Fig. 29.36c, the wire has been extended most of the way across the top of the circuit. Figure 29.36d shows the state of the circuit after the entirety of the top three rows have been read. Those rows are now completely overwritten with the wire itself, but the configuration of the original cells are stored in the repair circuit's storage elements ("Storage" in Fig. 29.35). Moreover, if there were errors in any of the configuration memories that were read, they should have been corrected before those configurations were stored.

In Fig. 29.37a, the wire has been retracted one step to the left, and the configurations of the original cells have been repaired in the wire's wake. In (b), more of the top rows have been repaired, as the wire continues to retract. Finally, in (c), the entire top three rows have been restored to their intended configuration. Moreover, since this operation occurs in parallel in each of the three circuits $C1$ $C2$ and $C3$, all three copies are repaired simultaneously.

In Fig. 29.38, the wire head has moved down three cells, and is ready to repeat the above steps on the next three rows of the circuit. This repeats until the entire 2D circuit has been repaired.

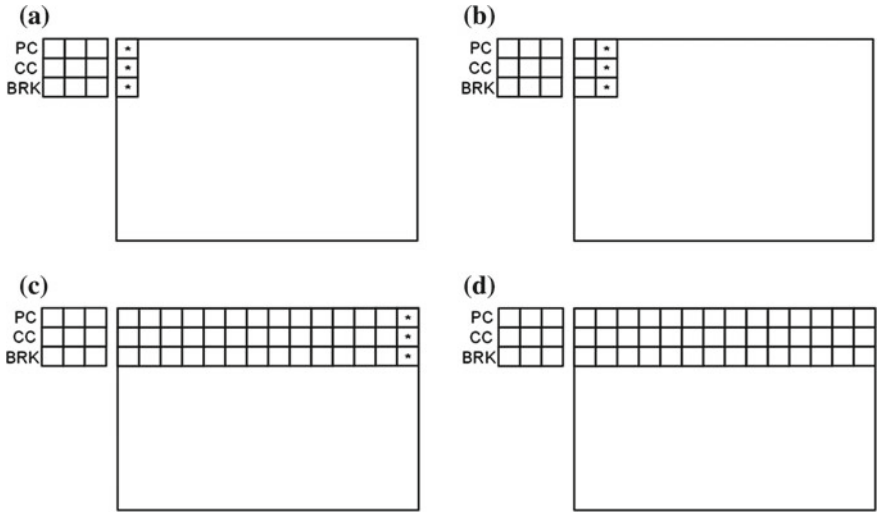

Fig. 29.36 Initial steps of cell repair. In figure **a** the cells marked "*" are about to be read. In **b** the original cells have been replaced with the wire, which is now ready to read the new cells marked "*". If **c** most of the top three rows have been read and replaced with wire. In **d** the entire top three rows have been read and replaced with wire. The overwritten cells' definitions are stored in the block marked "Storage" in Fig. 29.35

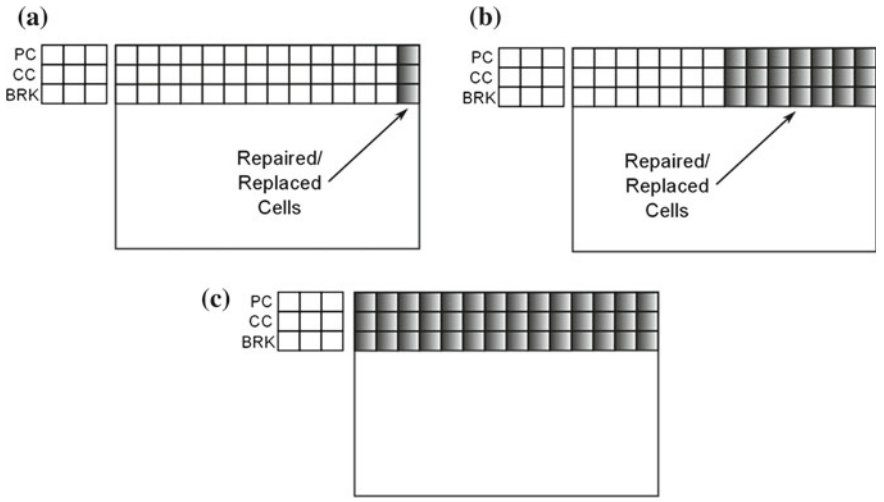

Fig. 29.37 Retraction of the wire. In figure **a** the *shaded* cells have been replaced with the original (if undamaged) or repaired versions of the original cells. In **b** the wire has been retracted further, with more of the original circuit replaced/repaired. In **c** the entire top 3 rows have been repaired/replaced

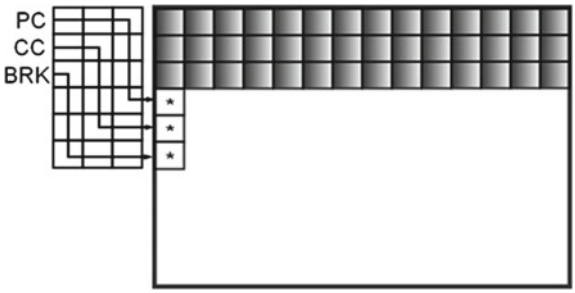

Fig. 29.38 After the top row has been replaced/repaired, the initial wire moves south 3 spots, and the above process repeats. This continues until the entire 2D region has been replaced/repaired

The above process works fine, provided that there is not an upset *in the same bit position* of the configuration memory for two or more cells *in the same position of the circuit*. For a 4-sided cell, each cell's configuration memory has 128 bits. Figure 29.39 shows a condition under which this scrubbing process will *not* work. In the indicated cell in circuit $C1$, one particular bit has an upset. If the same bit in the same cell in, say, circuit $C2$ also has an upset, then the majority vote on that bit will fail. Provided that condition does not occur, the scrubbing process will be able to restore the correct configuration.

In any system where circuitry is used to test or repair other circuitry, one must deal with the possibility that the test circuit itself may incur a defect. This present strategy, while not immune from this consideration, presents a relatively small opportunity for such critical defects: the wire heads, control and storage circuits must be defect-free. Of these, only the storage circuits have a size that increases with target circuit size: however, this is an order $O(\sqrt{n})$ situation: for an $n \times n$ target circuit, we need to protect a storage area whose size is $O(n)$ (since we are only processing one row of the target circuits at a time). For an extension of this technique to 3D circuits, the critical region's size is $O(\sqrt[3]{n})$.

One remaining issue is that while the scrubbing operation is underway, all 3 circuits are unusable. This may be acceptable, if the circuits are used infrequently, and the scrubbing is used to keep them clean in-between runs. An alternative approach is to keep 6 copies of the circuit, in two sets (A and B) of 3, and to alternate between a live circuit and a circuit-under-repair. A and B are each a complete fault-tolerant triple-redundant implementation of the target circuit, complete with its own copy of this scrubbing circuitry. While copy A is being used as a live circuit (with one of its three circuits pre-chosen to be the actual live circuit), copy B is being scrubbed. Once copy B is clean, it can become the live circuit, and copy A can be scrubbed. This alternation continues as long as the circuits are being used, and scrubbing occurs whether or not there are any errors present.

This technique thus combines self-analysis and self-modification to provide a reliable method of detecting and correcting defects in the configuration memories of a target circuit's cells. By continually detecting and repairing errors, the configuration memories are maintained in pristine condition; and by keeping them in this condition, they continue to be usable for detecting and repairing errors.

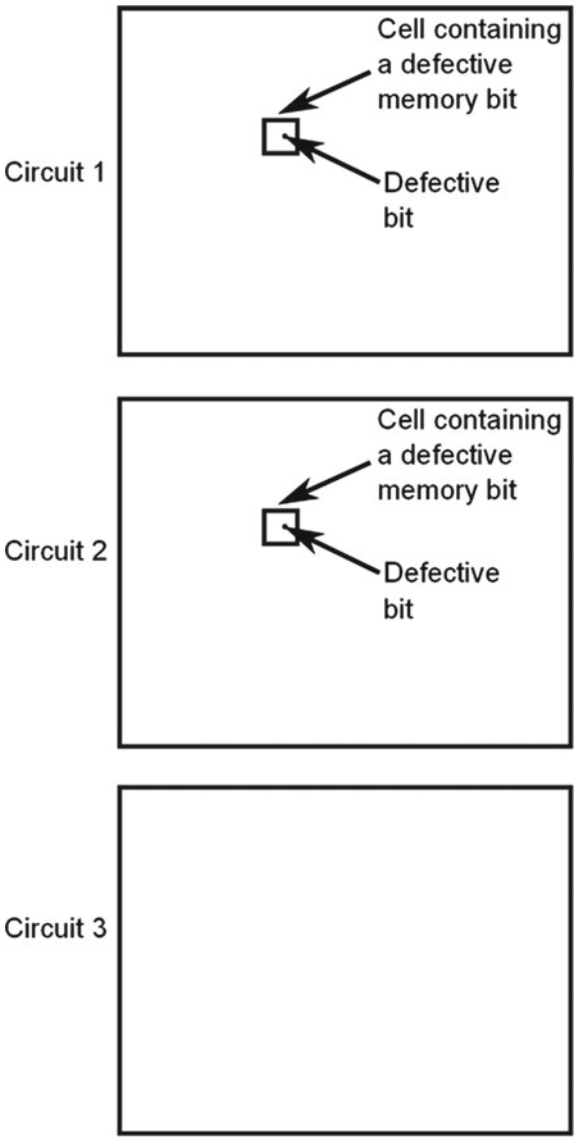

Fig. 29.39 A condition from which the system cannot recover. If a defect occurs in exactly the same bit of the same cell in two of the three copies of the circuit, the scrubbing mechanism will incorrectly modify the third copy to also be defective

29.4.3 Fault-Tolerant Detection and Isolation of Defects[2]

The Cell Matrix does not include any pre-existing circuitry for configuring a region of cells with a given set of configurations. While this may seem like a disadvantage of the architecture, its advantage over a hardwired configuration mechanism is that it

[2]This work was supported by NASA under contract NAS2-01049.

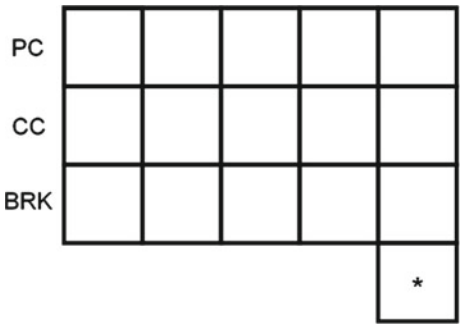

Fig. 29.40 Configuration of a single cell with a multi-channel wire

offers a great deal of flexibility in how the system is configured. This section explores two particular ways in which this flexibility can be used:

1. custom bootstrap circuits can be built for performing massively-parallel configurations; and
2. regions can be tested for defects before being configured.

Recall that a cell's configuration can be written by any adjacent cell (if that adjacent cell asserts one of the target cell's C inputs). Figure 29.40 shows a simple configuration example. "*" indicates the target cell which is being configured. This circuit employs a multi-channel wire: PC is used to send data to the target cell's D input; CC is used to assert the target cell's C input; and BRK is used to convey a secondary data signal. Circuitry located near the beginning of the wire (the end labeled "PC" "CC" and "BRK") is responsible for generating bitstreams into the PC and CC inputs so-as to configure the target cell appropriately.

Using this setup, the target cell can be configured to configure other cells in the vicinity, which allows the wire to be extended, bent, retracted, broken, and generally manipulated so as to allow the configuration of a **region** of cells near the initial target. Let us call the time required to configure a particular region of cells τ.

Figure 29.41 shows a similar setup, except that the channel now has two heads, and thus configures two target cells in parallel. In the setup shown, since the heads are situated 5 cells apart from each other, it's assumed a region no wider than 5 cells is being configured below each head. Note that aside from the extra propagation delay (arising from the longer path to the second target), **configuring two regions like this takes the same time as configuring one region**.

Figure 29.42 shows a further-modified setup (for simplicity, the multi-channel wire is now represented with a single line). In this case, there are multiple target cells (each at the end of an arrowhead). If there are n such heads, then n regions will be configured in parallel. While the configuration of these targets is thus relatively efficient ($O(1)$ for configuring n regions), one must consider the cost of building the wire itself, which is $O(n)$. Thus, to configure n regions is still an $O(n)$ process.

However, once an entire row of targets can be configured in parallel, it's possible to configure n regions in time τ. Those configurations can be used to create a new

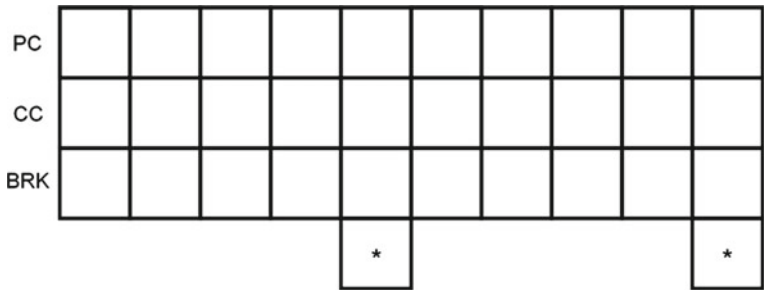

Fig. 29.41 A two-headed wire. The two target cells labeled "*" can each be configured at the same time

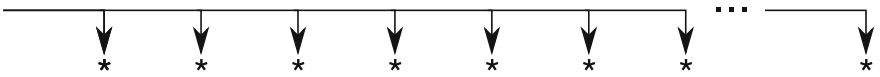

Fig. 29.42 A multi-headed wire. Each target cell "*" is configured at the same time

multi-headed wire below the initial wire, which can then be used to configure another n regions in time τ. Thus, in time τ we can configure one region; after $n\tau$ we can configure n regions; after $(n+1)\tau$ we can configure $2n$ regions, and so on. After $(2n-1)\tau$ we will have configured n^2 regions (Fig. 29.43).

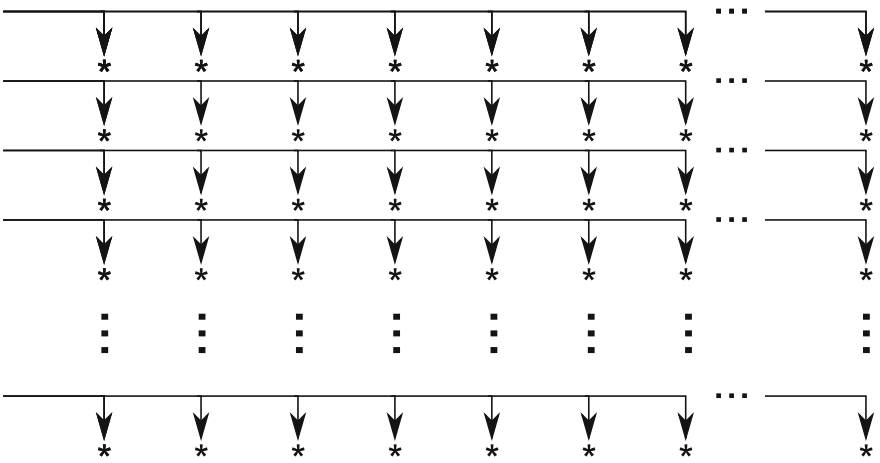

Fig. 29.43 Once an *initial row* has been configured (sequentially), it can be used to configure a *second row in parallel*. That row can configure a *3rd row*, and so on, with each row being configured in a fixed amount of time (independent of the length of the row). The entire region can thus be configured in $O(n^2)$ time

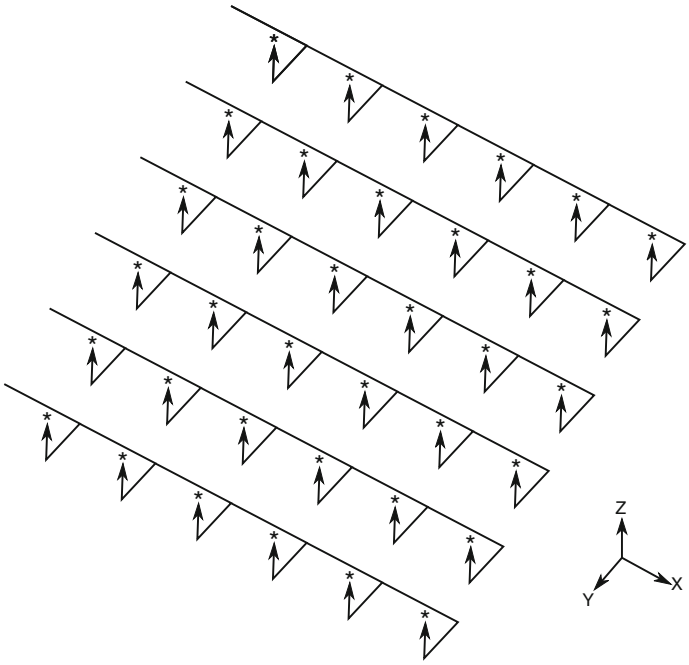

Fig. 29.44 Three-dimensional Medusa circuit. After configuring a plane of identical circuits, those circuits can configure a second plane in a fixed amount of time (independent of the dimensions of that plane). The entire 3D region can thus be configured in $O(n^3)$ time

This process can be further extended to support parallel 3D bootstrapping. If each configured region is built to allow configuration of cells in the Z axis, the circuit now allows configuration of n^2 regions in one additional timestep τ (Fig. 29.44). Target cells are again denoted with "*".

After another timestep τ, this newly-configured collection of n^2 circuits can be replicated again in the Z axis, creating another n^2 regions. Continuing for another $O(n)$ steps, and entire 3D collection of n^3 regions can be configured in $(3n - 2)\tau$ timesteps. Configuring n^3 regions thus takes a total time of $O(n)$ steps. Table 29.3 summarizes these results.

This is one example of a "*Medusa System.*" In general, a Medusa system is one that employs multiple heads to configure multiple regions in parallel, particularly with the goal of accelerating the configuration as it runs. The above example is a first introduction to a Medusa system, but a modified algorithm is actually more useful. By utilizing a different (but still accelerating) configuration order, we can incorporate fault detection and isolation into a parallel bootstrap system. Figure 29.45 shows the first 5 steps of this modified Medusa scheme. In this (and the next two figures), each square represents a collection of cells corresponding to a multi-cell region to be configured. At the end of the entire build process, all regions will be configured identically.

Table 29.3 Number of regions configured versus total configuration time

Number of Configured Regions	Total Time
1	τ
2	2τ
3	3τ
…	…
n	$n\tau$
$2n$	$(n+1)\tau$
$3n$	$(n+2)\tau$
$4n$	$(n+3)\tau$
…	…
n^2	$(2n-1)\tau$
$2n^2$	$(2n)\tau$
$3n^2$	$(2n+1)\tau$
…	…
n^3	$(3n-2)\tau$

The initial configurations proceed with linear time complexity, whereas the middle set of configurations accelerates to $O(\sqrt{n})$. The final set of configurations is $O(\sqrt[3]{n})$, which is also the time complexity for the entire process

The build begins with the configuration of a single region. The next 4 steps proceed as follows:

1. the first region configures a second region to its east;
2. those two regions configure two regions to their south (resulting in 4 configured regions);
3. the two westernmost regions configure two regions to their west (resulting in 6 configured regions); and
4. the three northernmost regions configure three regions to their north (resulting in 9 configured regions).

Continuing this way, after each pair of successive builds, the number of configured regions grows from k^2 to $(k+1)^2$. This is thus also an $O(\sqrt{n})$ process (and can again be extended into 3D for an $O(\sqrt[3]{n})$ process). Whereas the prior Medusa system grows first in one dimension, and then in the second, this system grows symmetrically from the middle, extending outward evenly in all directions. This is essential for incorporating fault handling into the system.

To detect faults, test patterns are applied in the region(s) which are being configured. Figure 29.46 shows a sequence of steps for parallel, fault-tolerant configuration of a 3×3 collection of multi-cellular regions. Steps which don't result in any new configurations (i.e., testing regions to the west from a westernmost cell) are not shown. In Fig. 29.46, the regions at the end of the arrowheads are being tested. Assuming the tested regions are found to be operating correctly, they are configured

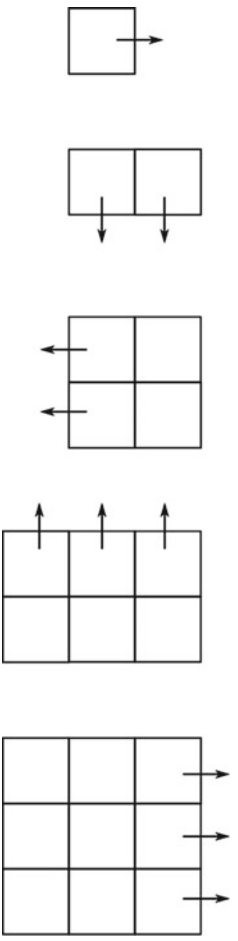

Fig. 29.45 First 5 steps of an increasingly-parallel build system. In each step, the arrow(s) show where the next build will occur. Note that all steps require the same amount of time to complete

as new additions to the Medusa system, and are subsequently used for further parallel test/configuration operations.

Figure 29.47 shows how this process runs in the presence of a fault. Here, a 3×3 layout of multi-cell regions is shown. The shaded regions have already been tested and configured; arrows indicate which adjacent regions are being tested (and, if found to be defect-free, configured). The region marked with a "*" contains a defect in at least one of its cells. Dark lines surrounding that defective region indicate *guard walls*, which are circuits within the adjacent defect-free regions that act to isolate the defective region from non-defective regions.

The following description concentrates on the detection and isolation of defective regions. To make use of this process for the construction of useful target circuitry, the target circuit is either built from elements contained within the testing circuitry

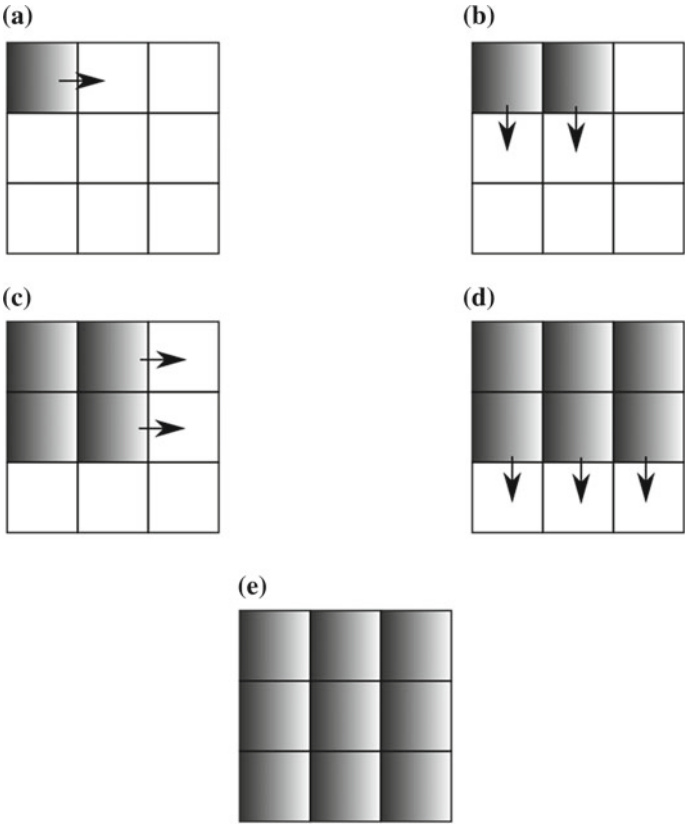

Fig. 29.46 Sequence of steps for parallel fault detection. Each square is a multi-cellular region. *Shaded* areas represent already-tested-and-configured regions. *Arrows* point to regions being tested and configured. In this example, there are no defects present in the region

itself, or a post-testing synthesis step reconfigures the collection of regions, using the test results to avoid defective cells.

The process begins with one initial region (in the upper-left corner) being configured. That region then tests the region of cells to its east ((a) in Fig. 29.47). Upon finding that region defect-free, it is configured as another piece of the collective test-circuit. Those two regions then test—in parallel—the regions to their south as shown in (b). The region on the left passes the tests, but the region on the right contains a defect, and fails one or more of the tests. A guard wall is activated in response to this, as illustrated by the heavy line to the north of the defective region in (c). The next tests—to the west and north—are not shown, as there are no cells to test in those directions.

In (c), non-defective regions now test to the east, resulting in the discovery of one good region (in the upper-right of the 3×3 area), as well as re-discovery of the defective region in the middle, causing a guard wall to the west of that region

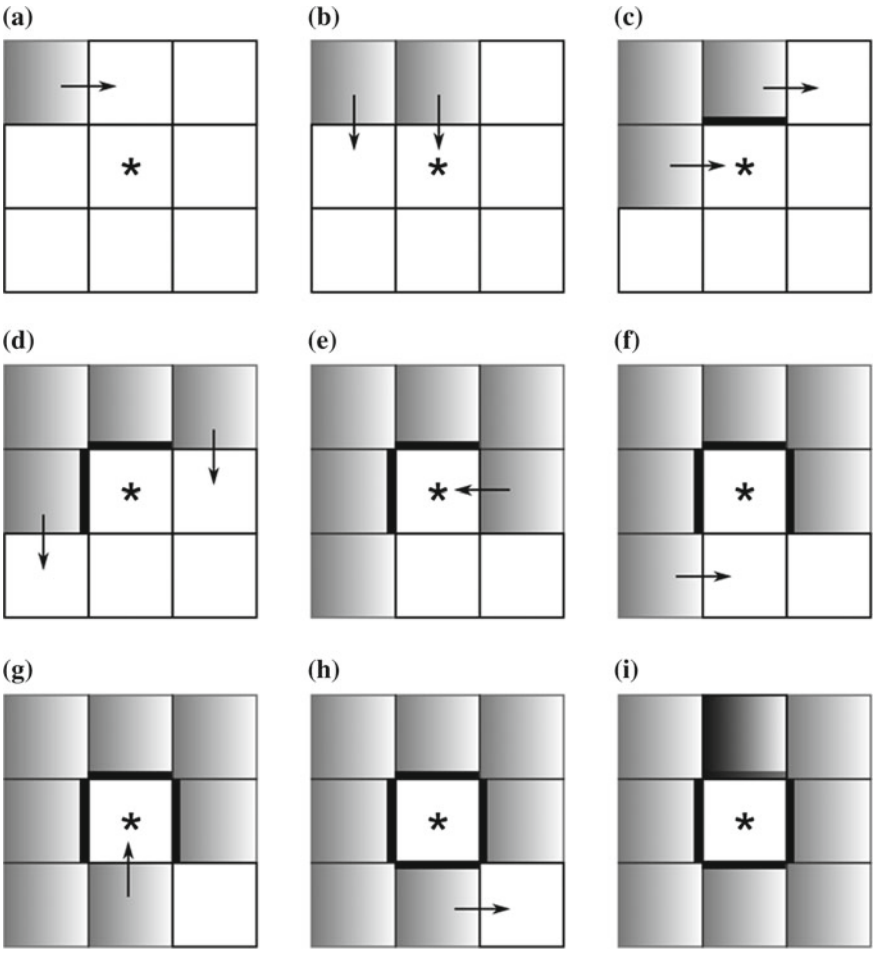

Fig. 29.47 Sequence of steps for parallel fault detection and isolation. Each square is a multicellular region. *Shaded* regions are already tested and configured. *Arrows* indicate which regions are being tested and configured. The region marked "*" contains a defect, which the system attempts to detect and isolate. *Bold* lines indicate guard walls, which work to isolate the defective region. Steps **a–i** show how the algorithm proceeds

to be activated (the heavy line in (d)). In (d), testing occurs to the south, resulting in configuration of two additional defect-free regions (seen in (e)). Note that the middle region of the upper row will attempt to re-test the defective region, but the guard-wall blocks all signals from entering or leaving that region, and thus those tests fail quickly.

In step (e), testing to the west *does* take place, with the only effect being the activation of a guard wall to the east of the defective region. The results of this are seen in (f), where the defective region is now surrounded on three sides by guard

walls. (f) also shows testing to the east, resulting in discovery of a 7th defect-free region, which in (g) will test to the north, and thus activate the final guard-wall around the defective region as shown in (h). Finally, one more test to the east leads to the final configuration shown in (i). As can be seen, the 8 defect-free regions have been identified and used in the establishment of a guard-wall surrounding the defective region in the middle.

As described above, this process as shown is only testing the 3×3 collection of regions, but in practice this is a preamble to subsequent synthesis of a parallel target circuit using these pre-tested regions. Further steps could then be used if differentiation of the regions is desired, as described in [12].

29.4.4 Detection of and Adaption to Cell-Level Orientation in a Disoriented Matrix[3]

One area of research for the physical assembly of a Cell Matrix is to employ *self-assembly* of individual cells into a larger 2D or 3D structure [15]. This raises a potential issue of *cellular disorientation*:

- each cell (in the 2D/4-sided case) has a notion of north, south, east and west;
- circuits are made from collections of cells communicating with each other;
- this means each cell needs to know how it is oriented relative to other cells.

For example, Fig. 29.48 shows a set of 4 cells being used to make a short wire. Each cell passes input from its west to its east, thereby making a 4-cell wire. In this case, each cell is oriented normally. In Fig. 29.49, the four cells are oriented randomly. While each has been configured to act as a wire from west to east, the cells are unable to act together to pass information from one cell to another. Given this orientation, three of the cells would need to be programmed differently, as shown in Fig. 29.50. The necessary equations are shown below each cell.

As described earlier, determining a cell's orientation is relatively easy. First, program the cell with a feedback circuit that copies west to west; then send a 1 out your own eastern output, and listen for an echo. If you receive it, then the target cell is oriented normally; if not, configure it to copy north to north, and try again. Knowing the configuration that results in an echo tells you the cell's orientation. Provided a cell's orientation can be determined, it is then easy to modify the cell's configuration string to restore the intended behavior. Thus, via introspection and appropriate modification of configuration instructions, a disoriented cell can be *effectively* re-oriented, so that it can be used as if it were oriented normally.

Of course, this only works for a target cell *directly adjacent* to a correctly-functioning set of cells. However (as usual), if an adjacent target cell can be effectively re-oriented, then it can be used to re-orient non-adjacent cells, by building and using

[3]This work was supported by the Cross-Disciplinary Semiconductor Research (CSR) Program award G15173 from the Semiconductor Research Corporation (SRC).

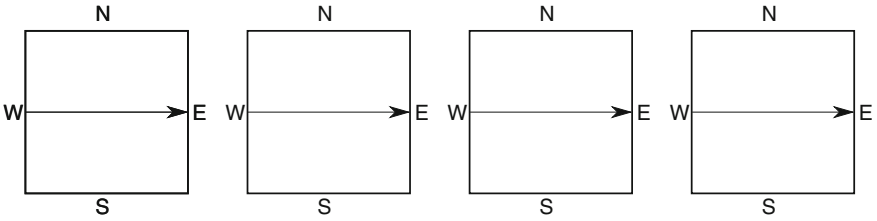

Fig. 29.48 Four cells, each configured to pass a single bit of data from west to east

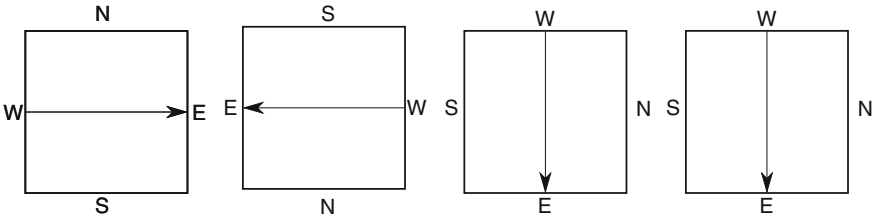

Fig. 29.49 Four cells, each configured to pass a single bit of data from west to east. In this case, the cells are disoriented, so the structure is not able to pass data from west to east

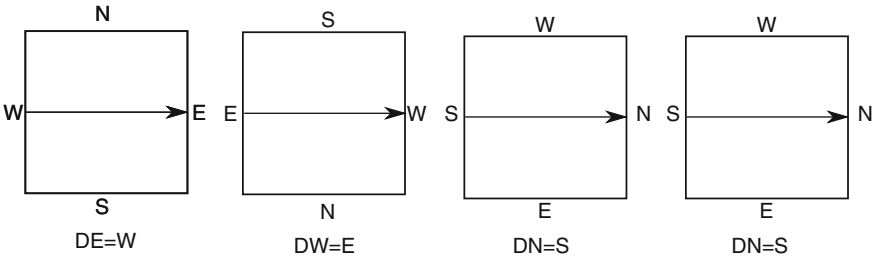

Fig. 29.50 By taking into account the disorientation of the cells, they can be configured differently to restore the data-passing capability of this circuit. Each cell's configuration is listed below the cell

multi-channel wires. The controlling circuit needs to be more complex, to allow for generation of test sequences and rotation of desired truth tables to match true orientation. Aside from that change, the build process for configuring a region of cells is essentially unchanged.

The situation is different though for parallel configuration. Consider the situation in Fig. 29.51. Here, a Medusa wire is setup to configure n cells (T_0 through T_{n-1}) in parallel. But unless all of these cells have the same orientation, there is no single configuration string that can be sent through the wire to configure all cells with the same effective configuration. For example, if T_0 is rotated 90 degrees clockwise and T_1 is rotated 90 degrees counter-clockwise, then the configuration strings sent into their respective D inputs must be adjusted accordingly. Similarly, to determine the orientation of a cell, test patterns must be fed to the cell and its behavior studied.

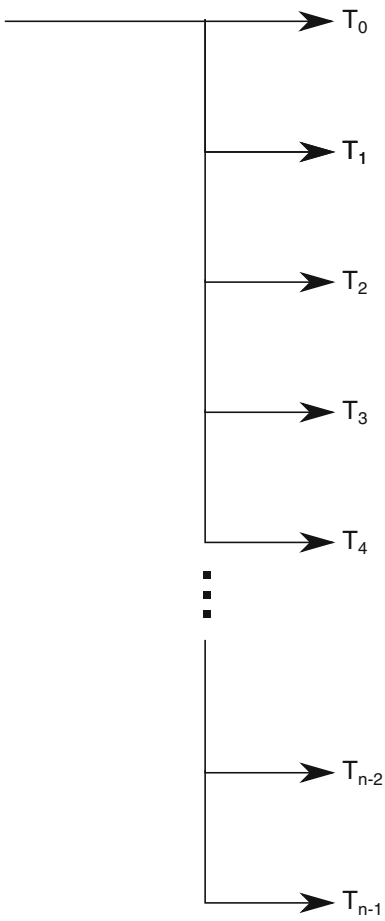

Fig. 29.51 A Medusa wire being used to configure n cells in parallel. To configure all target cells with the same *behavior*, different configurations need to be loaded into each cell based on its orientation

Since different cells may have different responses, each must be studied individually; yet to maintain parallelism, a set of cells should be tested in parallel. How can these seemingly conflicting requirements be reconciled?

The answer, again, is that the mechanisms of introspection and modification are purely local to each cell, meaning that the circuits to determine and respond to each target cell's orientation can be built near each target cell $T_0, T_1, \ldots, T_{n-1}$. This allows a universal set of instructions to be sent through the Medusa wire to each head, but also allows each head to act differently based on its local observations.

To accomplish this, each head is augmented with additional control circuitry as shown in Fig. 29.52. Three main channels are transmitted:

1. the PC, which delivers D inputs for configuring the target cell, as well as data to be used in testing a cell's orientation. The PC also returns data output from the target cell;

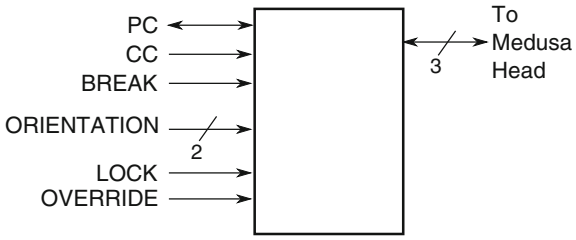

Fig. 29.52 Additional circuitry configured at each head of the Medusa wire. PC, CC and BREAK are the usual channels of the 3-channel wire. ORIENTATION carries two bits that code the current orientation being tested; LOCK is used to record the current value on the ORIENTATION lines; and OVERRIDE causes the CC to be asserted regardless of the ORIENTATION lines' values

2. the CC, which delivers the C input to control the target cell's mode (C or D); and
3. the BREAK line, which is an auxiliary data line that can be used to interrupt a wire and return it to a previous/shorter state.

In addition to these channels, there are three extra sets of lines sent to the control circuit:

4. an ORIENTATION wire, which carries two bits of information representing the current orientation being tested or configured;
5. a LOCK signal used to indicate that a test pattern is being delivered; and
6. an OVERRIDE signal, used to set the target's C input regardless of the state of the ORIENTATION lines.

Figure 29.53 shows how these lines work together to solve the simultaneous re-orientation problem for multiple heads. Asserting the OVERRIDE line causes the CC line to be passed to the target cell's C input, thus allowing the target to be configured regardless of its orientation. This is used to load a feedback test pattern (e.g. $DW = W$) into the target cell. Next, the ORIENTATION lines are set to 00 (indicating a rotation of 0 degrees from "normal" orientation), and a 1 is then sent through the PC channel. While maintaining those signals, the LOCK line is momentarily pulsed high. Assuming the target cell is oriented normally, the two D flip flops will load the current orientation (00); otherwise, the flip flops' values will remain unchanged.

By cycling through all four possible test patterns ($DW = W$, $DN = N$, $DE = E$ and $DS = S$), the flip flops will be loaded with the orientation that resulted in feedback detection, and thus indicate the orientation of the target cell. This completes the introspection phase of the system. Next, this information is used to configure the target cell with the desired configuration. This is accomplished by asserting the CC line and sending the desired configuration four times: once for each possible orientation. As each configuration is sent, the ORIENTATION lines are set to indicate the orientation corresponding to the current configuration being sent. In the unique case where the ORIENTATION lines match the saved orientation, the flip flops' MATCH output will be asserted, which will allow the CC signal to drive the target cell's C input.

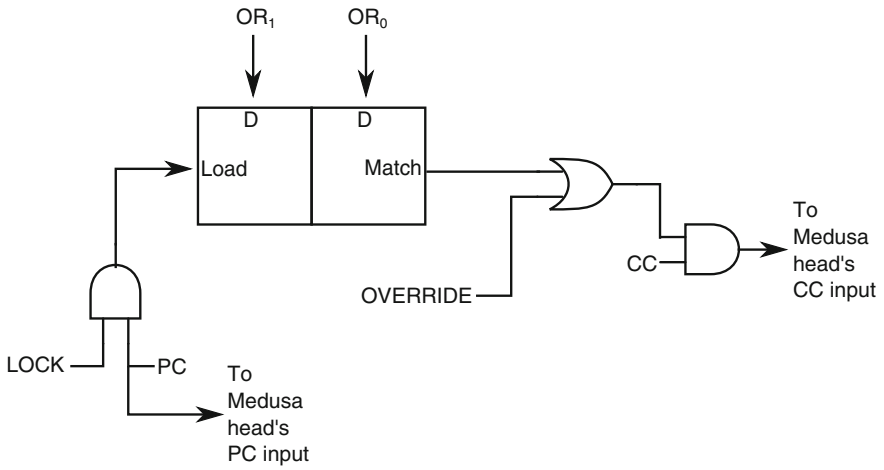

Fig. 29.53 Circuitry for simultaneous determination and correction of head cell orientation. LOCK is used to load the 2-bit orientation latch with the current orientation (OR_1 and OR_0) provided PC is receiving a 1 (echo) from the target cell. If the orientation matches the latched value, the MATCH output is asserted, which causes CC to be routed to the Medusa head. The OVERRIDE signal also causes CC to be routed to the Medusa head

Since the control circuitry in Fig. 29.53 is local to each head of the Medusa wire, the saved orientation information is correct for each target cell, and after sending the four possible configuration strings, each target cell will have been configured as desired. While this now takes $4\times$ longer than configuring a normally-oriented matrix, the process is still $O(\sqrt{n})$. The $4\times$ factor can be eliminated (in exchange for a $4\times$ increase in space utilization) by making the Medusa Wire's PC channel four times as wide, so that the configuration bitstream for all four possible orientations are sent in parallel. The control circuit can be modified to act as a 4-1 selector to route one of those four PC sub-channels into the target cell's D input.

Thus, by combining analysis and self-modification with local hardware to preserve and utilize local information, a high degree of parallelism can be maintained despite the need to perform different operations at each Medusa head.

29.5 Conclusions and Future Work

The ideas of introspection and self-modification have been discussed, with respect to a particular self-configurable architecture called the Cell Matrix. Native mechanisms for support of introspection and self-modification have been described, and these have been used to develop circuits and methodologies for various low-level behaviors, including wire building, parallel configuration, defect detection and isolation, orientation-determination, and self-location within the matrix. These have

been used as building blocks to create more-complex systems that utilize introspection and self-modification in a variety of ways. The simplest example demonstrates circuit synthesis in response to system state; whereas the most complex uses parallel differentiation in response to local introspection to tailor circuit synthesis as a way to work with disoriented cells.

The techniques and examples presented herein are starting points for more-sophisticated systems. It is hoped that this work will be extended in multiple directions, including work in higher-dimensional spaces; extension to other self-modifying, introspective architectures; and creation of larger-scale end-products that exhibit fault detection/avoidance/adaption, parallel synthesis and operation, and self-analysis. Eventually, these may not be ends unto themselves, but will simply be pieces of a larger picture: a picture of artificial systems that utilize self-awareness to afford resiliency, adaption and growth as part of their normal operation.

References

1. Birkner, J., Chua, H.T., Chan, A.K.L., Chan, A.: Programmable logic array with added array of gates and added output routing flexibility. US Patent 4,758,746, 19 July 1988
2. Ceschia, M., Violante, M., Sonza Reorda, M., Paccagnella, A., Bernardi, P., Rebaudengo, M., Bortolato, D., Bellato, M., Zambolin, P., Candelori, A.: Identification and classification of single-event upsets in the configuration memory of sram-based fpgas. IEEE Trans. Nucl. Sci. 50(6), 2088–2094 (2003)
3. Dubowski, S.: Software defined radio aims to bury gsm-cdma hatchet. Netw. World Can. 12(17) (2002)
4. Durbeck, L.J., Macias, N.J.: Defect-tolerant, fine-grained parallel testing of a cell matrix. In: ITCom 2002: The Convergence of Information Technologies and Communications, pp. 71–85. International Society for Optics and Photonics (2002)
5. Franzoi, S.: Social psychology (1996)
6. http://quotes.lifehack.org/quote/carl-jung/your-vision-will-become-clear-only-when/. Accessed 30 June 2015
7. Jamison, C.: Finding Happiness: Monastic Steps for a Fulfilling Life. Liturgical Press (2009)
8. Kuon, I., Tessier, R., Rose, J.: Fpga architecture: survey and challenges. Found. Trends Electron. Des. Autom. **2**(2), 135–253 (2008)
9. Lyons, R.E., Vanderkulk, W.: The use of triple-modular redundancy to improve computer reliability. IBM J. Res. Dev. **6**(2), 200–209 (1962)
10. Macias, N.J.: Circuits and sequences for enabling remote access to and control of non-adjacent cells in a locally self-reconfigurable processing system composed of self-dual processing cells. US Patent 6,297,667, 2 Oct 2001
11. Macias, N.J.: Self-Modifying Circuitry for Efficient, Defect-Tolerant Handling of Trillion-element Reconfigurable Devices. Ph.D. thesis, Virginia Polytechnic Institute and State University (2011)
12. Macias, N.J., Athanas, P.M.: Application of self-configurability for autonomous, highly-localized self-regulation. In: Second NASA/ESA Conference on Adaptive Hardware and Systems, 2007. AHS 2007, pp. 397–404. IEEE (2007)
13. Macias, N.J., Durbeck, L.J.K.: Self-assembling circuits with autonomous fault handling. In: Proceedings. NASA/DoD Conference on Evolvable Hardware, 2002, pp. 46–55. IEEE (2002)
14. Macias, N.J., Durbeck, L.J.K.: Self-organizing computing systems: songline processors. In : Advances in applied self-organizing systems, pp. 211–262. Springer (2013)

15. Macias, N.J., Pandey, S., Deswandikar, A., Kothapalli, C.K., Yoon, C.K., Gracias, D.H., Teuscher, C.: A cellular architecture for self-assembled 3d computational devices. In: 2013 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH), pp. 116–121. IEEE (2013)
16. Patwardhan, J.P., Dwyer, C., Lebeck, A.R., Sorin, D.J.: Circuit and system architecture for dna-guided self-assembly of nanoelectronics. In: Foundations of Nanoscience: Self-Assembled Architectures and Devices, pp. 344–358 (2004)
17. Sekanina, L., Dvok, V.: A totally distributed genetic algorithm: From a cellular system to the mesh of processors. In: Proceedings of 15th European Simulation Multiconference 2001, pp. 539–543 (2001)
18. Udall, J., Teuscher, C., Macias, N.: Truncated Octohedron Circuits. private communication (2015)
19. Zienkiewicz, O.C., Leroy Taylor, R.: The Finite Element Method, vol. 3. McGraw-Hill, London (1977)