1

Self-Organizing Digital Systems

Nicholas J. Macias and Lisa J. K. Durbeck

Cell Matrix Corporation, Blacksburg, Virginia, USA

1.1 Introduction

Theory is at the threshold of understanding how to translate self-organizing principles and processes to human-formed systems. However, practice lags behind theory. This chapter endeavors to provide inroads into the application of self-organization principles to one aspect of electronics systems, namely, digital logic.

Digital circuitry proliferated from the early transistor-Transistor Logic (TTL) circuits of the 1960's to the now mass markets of computers, mobile phones, T.V.s, and numerous other consumer products. The fundamental component of digital circuitry is the logic gate from which complex functions can be derived and explained with the use of digital logic. Due to its widespread use and complex application, digital logic is arguably a good target for applying concepts from self-organizing systems.

The ultimate goal of applying self-organization concepts to digital logic is to devise theory and practice as to how digital logic could be constructed and operated as a self-organizing system. Our approach has been to devise reconfigurable logic hardware and an architecture that permits self-organizing processes, and then to begin methodically developing self-organization concepts and their translation to practice within this framework. This work requires changing the way digital logic is both designed and built, providing so-called *primitives*, or fundamental behaviors, for self-organizing systems, along with a way to build upon these primitives to conceive of, compose, and orchestrate self-organized digital logic.

To achieve an inherently self-organizing infrastructure, a number of departures from conventional digital logic design are required. These include:

- reworking how the system is controlled by placing the control within the componentry of the system itself, that is, if the system is composed entirely of digital logic, then its digital logic must have the ability to control digital logic, creating a new class of digital logic that is able to dynamically change, and able to modify both itself and its neighbors;

- the need to incorporate this self-inspecting, self-modifying power into systems without again introducing the hierarchy of controller and controlled, so that they are both loosely onto each other, or interchangeable; and
- the need to conceive of and develop strategies that build upon these core capabilities to re-conceive system monitoring and control as a distributed process largely enacted within the confines of the system itself, and composed of many simple, localized activities with significant autonomy in their ability to decide and act on local information.

The work described below will further ground this discussion of objectives and insights with concrete examples as to how these properties are included in the hardware architecture we are developing, and gives some insight by example as to how these simple, foundational or underlying processes can be used to compose digital logic that is self-organizing and dynamically self-modifying.

As a new approach digital logic design, it is unlikely that we have developed all the primitives necessary for every class of problem. We therefore anticipate that as we apply this work to more classes of problems, the need for other primitives is likely. We have also not yet developed the full set of useful mid-level behaviors, built from the primitives, that are likely to be necessary for self-organizing digital logic designs. However, we report here on the current state of the art and outline areas in which this work will be further applied.

Several separate aspects of digital logic production may benefit from the application of principles of self-organization, both in the structure and function of digital logic circuits. In the case of an FPGA , a self-organizing process could be used to fabricate the physical hardware; the primitive functions of the hardware could use and enable self-organization; and any logical level could do the same for the logical layer above it by supplying primitives that the upper layers can employ. In this chapter we present research done on the design of the FPGA and its low level logic behavior to develop self-organizing primitives that can be used to structure the logical levels above it, or can be invoked by those upper logical levels. We view this as foundational work toward the eventual integration of self-organizing behavior into digital logic.

A key aspect of design at the hardware architecture and low-level system structure level is the data path and the control path of the computational architecture. Much of the discussion below will refer to the system *control*. This should be understood to be all those processes that direct the operation of the overall system, such as those scheduling activities or processes, synchronizing the actions of disparate processes, and maintaining the overall system and all subprocesses of it in proper working order.

System maintenance is currently done largely by people rather than processes on the machine because of the need for physical action in many cases such as the replacement of failing memory cards and disk drives. However, in concept, the act of inspecting the equipment for failure can be integrated into the design of the processes that run on the equipment. When systems scale upward to the point that they contain 10^{17} or more logical devices, there will likely be sufficient incentive to reorganize hardware inspection as a distributed, localized process as well, on account of the unavoidably high frequency of hardware upsets. Similarly, the process of inspecting the initial constructed hardware for defects may also become tedious enough that it has

similar incentive to reorganize hardware inspection as a distributed, localized process running on the hardware itself.

To invest the low level architecture of digital logic systems with properties of self-organizing systems, it appears to be critical to change the typical computer control path into one that is based instead upon strictly local interactions, and to then conceive of the overall control path as being a complex distributed process that emerges out of many local control actions. Our work provides this new kind of control path, and we detail a number of examples of how it is used to recast control as a highly localized process, and then describe examples in which we have built up increasingly complex systems that are composed out of these small, highly localized processes. Management of the actions of processes is thus transformed from the typical centralized control of a manager that is making decisions and controlling the system outside the system itself to distributed management and control.

Self-organization may be an antidote to the fact that the complexity of controlling and managing systems has been at least proportional to the size of the system, the number of components under management. Managing 10^{18} components using traditional methodology does not appear to be a tenable proposition. A hypothesis that underlies the work presented here is that approaches to deal with the complexity of a very large system likely require at least an equally large system as the manager, and that it may be preferable that the manager be not separate but integral with the system under management, since, for example, the details requiring management decisions come down to a very large number of small details that may be extremely difficult to output every nanosecond. Achieving this co-functioning of manager and process under management appears possible with a particular type of infrastructure to the underlying system, one that combines sufficient flexibility with an inherently self-referential structure that makes introspection and autonomous self-modification feasible. The architecture presented here has the necessary properties to test this hypothesis. Section 1.2 will describe a particular system called the Cell Matrix (Macias 1999; Durbeck and Macias 2001d) that possesses these necessary characteristics.

1.1.1 Background on The Concept of Self-Organization

The concept of self-organization has application to a number of domains. In the domain of living systems, self-organization is a central theme in many areas (Darwin 1859; Kauffman 1993). Philosophical considerations date back even further (Haldane 1931; Lennox 2001). In the domain of artificial systems, many facets of self-organization have been explored, including autonomous behavior and self-repair (Aspray and Burks 1987).

Approaches to self-organization can be grouped into at least two main categories: one we will call a *statistical* approach, and the other an *engineered* approach. Statistical approaches seek to manage complexity by *discovering* algorithms or techniques. Such approaches are not necessarily concerned with *how* the given task is accomplished, only with how well it is presently being accomplished. Examples of statistical approaches include neural networks (e.g., (Abdi 1994)) and genetic algorithms (e.g., (Koza 1992)). Genetic algorithms have been extensively applied to the development

of electronic circuits, in a field called Evolvable Hardware (e.g., (Thompson 1996)). It should be noted that much of this work draws inspiration from biology (Darwin 1859).

In contrast to statistical approaches, engineered approaches seek to more-deliberately achieve some set of goals by following more of a pre-defined algorithm. Interestingly, many engineered approaches also draw inspiration from biology, including the Electronic Embryology (Embryonics) work of LSL (Prodan et al. 2003; Ortega-Sanchez et al. 2000), and the Supercell work of Cell Matrix Corporation (Macias and Durbeck 2004). The discussion in this chapter falls into the domain of the engineered approach.

1.1.2 Chapter Organization

The remainder of this chapter is organized as follows: Section 1.2 will describe in detail a particular target processing architecture called the Cell Matrix, which possesses an inherently self-organizing infrastructure; Section 1.3 will describe simple examples of self-modifying circuitry on the Cell Matrix, which will form the building blocks for larger-scale self-organizing systems; Section 1.4 will discuss such larger systems; and Section 1.6 will conclude with discussions of implementation details, including manufacturing and CAD issues related to the Cell Matrix.

1.2 Target Platform: The Cell Matrix

This chapter discuses the distributed management and control of electronic circuitry implemented on a specific reconfigurable platform called the Cell Matrix (Macias 1999; Durbeck and Macias 2001d). While there are a number of commercially-available reconfigurable devices (*Field Programmable Gate Arrays*, or FPGAs), including many from Xilinx, Inc., most available devices are essentially externally-controlled, i.e., they require intervention from an outside system (e.g., a PC) in order to be configured or re-configured (Xilinx, Inc. 2006). Moreover, even among devices that can hold several simultaneous configurations (Trimberger 1998), those configurations are generally pre-created, externally, again using a PC or other extra-FPGA system. In contrast, the Cell Matrix is fundamentally an *internally*-configured device: the configuration of each cell is written–and read–by those cells connected to it, its immediate adjacent neighbors . Only cells situated on the perimeter of the matrix (which are thus missing one or more neighbors) are accessible from outside the system. This is fundamentally different from most other devices, where typically *every* cell can be accessed from outside the system by simply sending a long configuration string throughout the device.

While it may seem unusual, and perhaps disadvantageous, to have such limited access to cells from outside the system, this is in fact a critical characteristic of the Cell Matrix, and is directly linked to its ability to implement autonomous, self-organizing circuitry. Having local-only cell control also allows the system to scale , without specific regard for scaling the control structures.

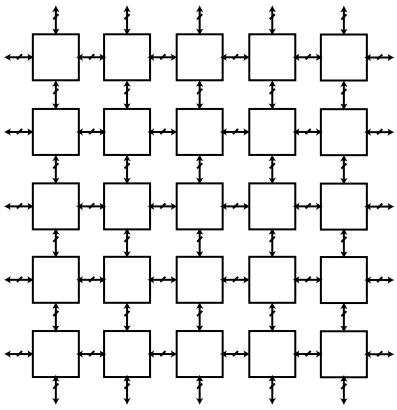

Fig. 1.1. 5×5 Collection of Two-Dimensional, Four-Sided Cell Matrix Cells

1.2.1 Basic Cell Structure

A Cell Matrix is a regularly-tiled collection of simple reconfigurable elements called *cells*. These cells are arranged in a fixed, identical topology throughout the matrix, and that topology defines a notion of a cell's *neighbors*: the neighbors of a cell "X" are all those cells that are immediately connected to X. Each cell receives a single input bit (called its "D Input") from each of its neighbors, and generates a single output bit (its "D Output") to each of those neighbors. Figure 1.1 shows a two-dimensional collection of four-sided cells. In this topology, each cell has four immediate neighbors. We will mainly be discussing two-dimensional, four-sided cells in this chapter. However, (useful) two-dimensional cells can have as few as three sides, or may have more than four, though four is the most typical number. Cells can also be three-dimensional, having as few as four sides, but more typically six. Higher-dimensional cells are also possible, though anything higher than three dimensions ceases to be (architecturally) infinitely-scalable because there is no way to organize the cells topologically that puts all neighbors a finite, very small distance from each other.

1.2.2 Cell Structure

Each cell contains a small memory which stores a *truth table*. The truth table maps input combinations to outputs: given the set of incoming bits from all of a cell's neighbors, the cell's outputs are precisely determined by the information in the cell's truth table. This mechanism allows a single cell to implement simple combinatorial functions, such as basic logic gates, single-bit adders or multiplexers. Cells can also act as simple wire: a block for passing data from one side of itself to another — or, viewed differently, a block for allowing two non-adjacent cells to share data with each other. Implementing wires is a major use of cells.

1.2.3 Cell Configuration

The act of loading truth table information into a cell is called *cell configuration* or "configuring a cell." Similarly, the act of loading truth table information into a number of cells is called "Configuring the Cell Matrix." While single-cell circuits are necessarily extremely simplistic, configuring a group of cells appropriately leads to multi-cell circuits, which can be arbitrarily complex. Because single cells can implement any fixed input-to-output mapping, and cells can be interconnected via intervening cells, any circuitry that can be implemented using traditional digital circuit design can also be implemented on a Cell Matrix.

Figure 1.2 shows a more-detailed view of a single cell. As can be seen, each side has two input lines and two output lines, which connect it to each of its immediately-adjacent neighbors. One line is labeled "D" and the other "C." The D inputs are used to select information from the cell's truth table, and the D outputs are set based on the truth table's values, as described above. *But this is the case only if all the C inputs are 0.*

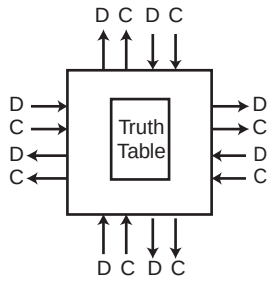

Fig. 1.2. Four-Sided Cell Matrix Cells. Each neighboring cell reads and writes two bits (D and C). C inputs determine the **mode** of the cell. D inputs either select C and D output values from the Truth Table (in D mode) or supply new values for the Truth Table (in C mode).

When all C inputs are 0, the cell is said to be in "D" mode. If, however, any C inputs are set to 1, then the cell is in "C" mode. C mode is the configuration mode of a cell: it is the mode in which a cell's truth table can be modified. In C mode, a cell's truth table can also be examined. A cell that is asserting one of its own C outputs, and is thus asserting a neighboring cell's C input, is able to read and write that neighboring cell's truth table. Note that if more than one C input is set to 1, then the cell's truth table is sent to multiple neighbors, and its new truth table is determined by a combination (logical ORing) of its neighbors outputs.

This C-mode operation is like the *unit measure* of self-organization for the entire architecture; using it, cells are configured by a neighboring cell. The extreme locality of this operation makes the entire architecture fine-grained in its reconfigurability, and makes configuration a distributed, local process.

Cell configuration is the only inherently clocked operation in the Cell Matrix: a single system-wide clock is used to serially shift out the current contents of a cell's

truth table, and to serially shift in new truth table bits. These bits are read-from and written-to the D output and input lines, respectively, on the same side on which the cell's C input is asserted (called the *active side*). Figure 1.3 shows an example of adjacent-cell interactions in D and C modes. In Figure 1.3.a, Cell Y is in D mode, since all of its C inputs are 0. It thus uses its four D inputs to select a single row in the 16×8 truth table memory, and sends the selected eight output values to its eight outputs (four C and four D).

In Figure 1.3.b, one of Cell Y's C input is asserted (the one supplied by Cell X). This places Cell Y into C mode, the mode in which its truth table is read and written. Each time the system-wide clock ticks, the D input supplied by Cell X is loaded into Cell Y's truth table, at a position that changes with each tick (in Figure 3b, the bit in the third row, second column is being written). Additionally, the previous value stored in that location is made available on the D output to Cell X. All other D outputs from Cell Y are forced to 0, as are **all** of Cell Y's C outputs (so that a cell being configured cannot itself simultaneously configure another cell). By convention, if two or more of a cell's C inputs are asserted, the bit value loaded into the cell's truth table is the logical OR of the D inputs on all active sides.

Using this simple interaction scheme, it is possible for any cell to read and write any neighboring cell's truth table. Since cells along the edge of the matrix have some of their inputs and outputs unconnected, as shown in Figure 1.4, those edge cells can be configured from outside the matrix, if their C and D inputs (and, perhaps, outputs) are made available. In Figure 1.4, all edge cells have their inputs and outputs accessible from the edge of the matrix on at least one cell side (corner cells are accessible from two sides).

Figures 1.5-1.7 show the full details of a cell configuration operation: Figure 1.5 shows a single cell configured as a NOR gate; Figure 1.6 shows the cell's corresponding truth table; and Figure 1.7 shows a timing diagram for configuring the cell. The cell is first placed into C mode by raising one of its C inputs. As soon as the cell enters C mode, the current value of the cell's first truth table bit is sent to the corresponding D output (not shown in the figure). On the next rising edge of the system clock, the D input is sampled and latched. On the next falling edge, the latched value is loaded into the truth table, and the current truth table's next bit is sent to the D output. Note that this timing makes the truth table's current bit values available on the D output half a cycle before the new bit value must be presented to the D input. This makes it simple to read a truth table bit and then re-write the same bit, thus performing a non-destructive read.

Before the 4^{th} clock tick, the D input is raised. This "1" value is latched/loaded into the cell's truth table on the next rising/falling edge of the system clock. Similarly, a "1" is loaded during the 12^{th} cycle following the cell's entry into C mode. All other incoming bit values are 0.

A few cycles later, the cell's C input is set to 0, and the cell returns to D mode. Assuming its truth table initially contained all 0s, its truth table is now as shown in Figure 1.6.

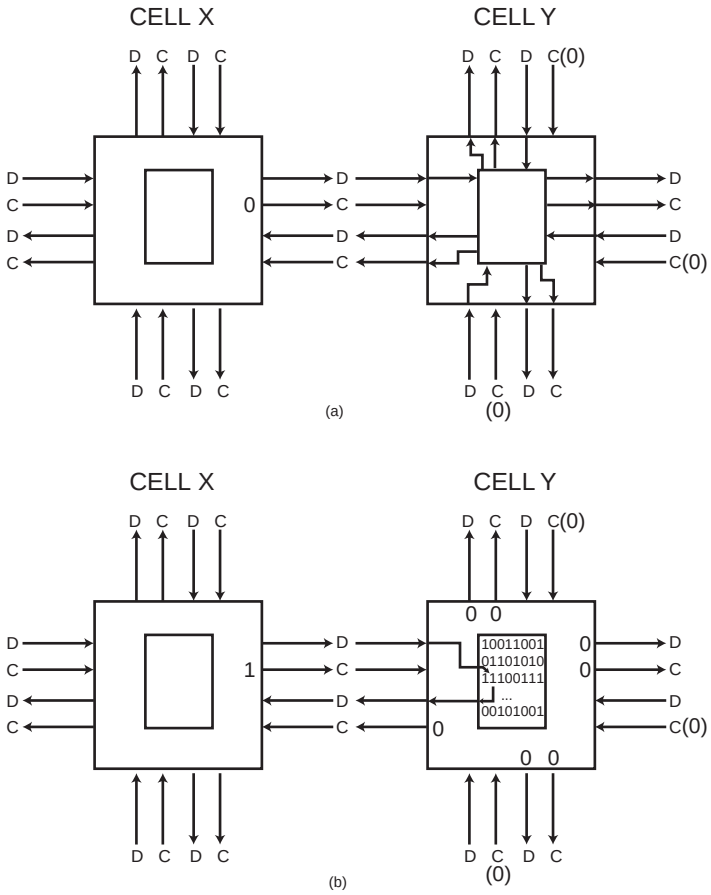

Fig. 1.3. The Two Mode of Cell Operation. In (a), cell Y is in D Mode (all C inputs are 0). Its four incoming D values are used to select 8 output values from its truth table. Those output values are sent to the cell's 8 output lines (4 C lines and 4 D lines). In (b), cell X is asserting a 1 to one of cell Y's C inputs, and thus cell Y is in C-mode. In this mode, the D input from Cell X supplies new values for Cell Y's truth table. Each time the system clock ticks, a new incoming bit value is sampled, and loaded into Cell Y's truth table. Cell Y's current truth table bits are simultaneously sent out the D output to cell X. In the figure above, the second bit in the third row of the truth table is being read and written by Cell X. All of Cell Y's other outputs (C and D) are forced to 0.

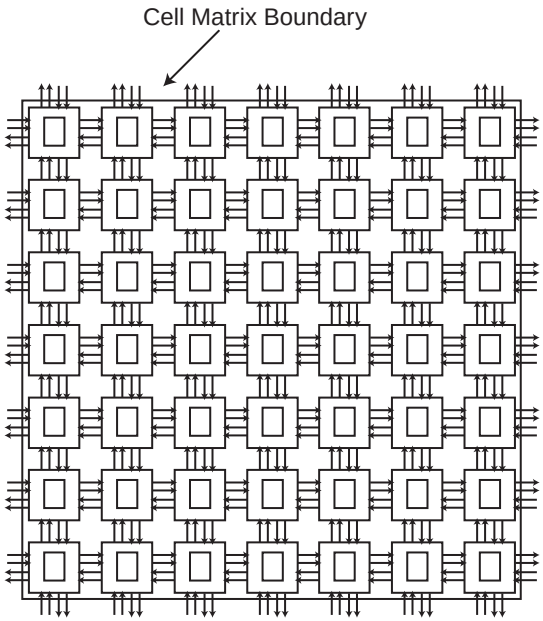

Fig. 1.4. 7×7 Cell Matrix. Edge cells have their D and C inputs and outputs accessible from outside the matrix. Corner cells have I/O accessible on two of their sides.

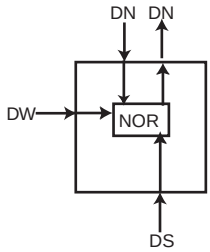

Fig. 1.5. Single Cell Implementing a Three-Input NOR Gate.

1.2.4 Self-Configuration

Because cells are able to read and write other cells' truth tables, the Cell Matrix can be configured from *inside* the Cell Matrix itself. This makes the Cell Matrix a *self-configurable* system, i.e., circuits can be constructed that read and write cell configurations, and thus can analyze and change circuitry within the matrix. Circuitry constructed on the Cell Matrix can process data that represents logical values, characters, integers, floating point numbers, or any sort of data structure. But additionally, circuitry constructed on the Cell Matrix can also process a unique type of data: circuit configuration information. And because the mapping between circuit configuration and circuit behavior is very straightforward, one can construct circuits that effectively *process other circuits.*

INPUTS				OUTPUTS							
DN	DS	DW	DE	CN	CS	CW	CE	DN	DS	DW	DE
0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	1	0	0	0	0	1	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0	0

Fig. 1.6. Truth Table Corresponding to Fig. 1.5

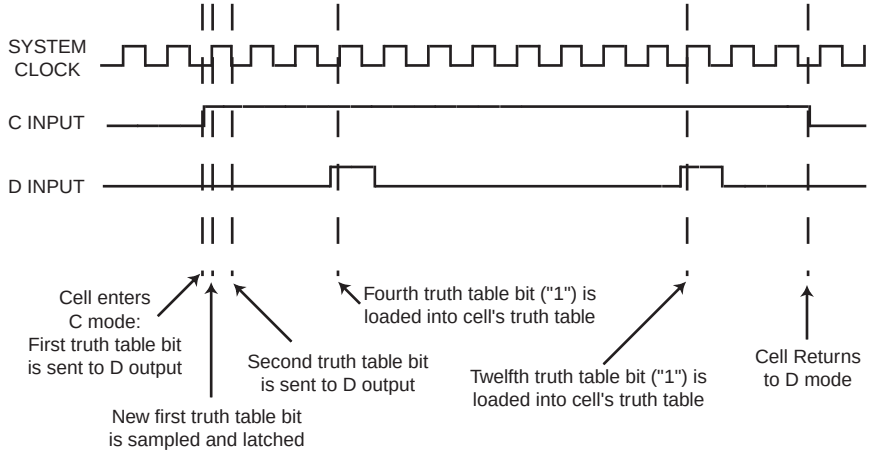

Fig. 1.7. Truth Table Programming Sequence. After cell is placed into C mode, a "1" bit is loaded on the 4^{th} and 12^{th} ticks of the system clock. The cell is returned to D mode two ticks later. This loads 14 bits into the cell's Truth Table: 0001 0000 0001 00.

Moreover, there is no hardware-level or architectural difference between a cell that is being configured and the one that is configuring it. Figure 1.8 shows some of the possibilities resulting from this fact. In Figure 1.8.a, Cell X, Cell Y and Cell Z are all in D mode, since their C inputs are all 0 (all inputs are assumed to be 0 unless shown otherwise). Each cell is simply receiving D inputs, using them to address their internal truth table, and producing D and C outputs accordingly.

In Figure 1.8.b, Cell X is asserting a 1 on its C output to Cell Y. This places Cell Y into C mode. In this mode, Cell Y's truth table is being configured by Cell X, by sampling the D outputs sent from Cell X to Cell Y.

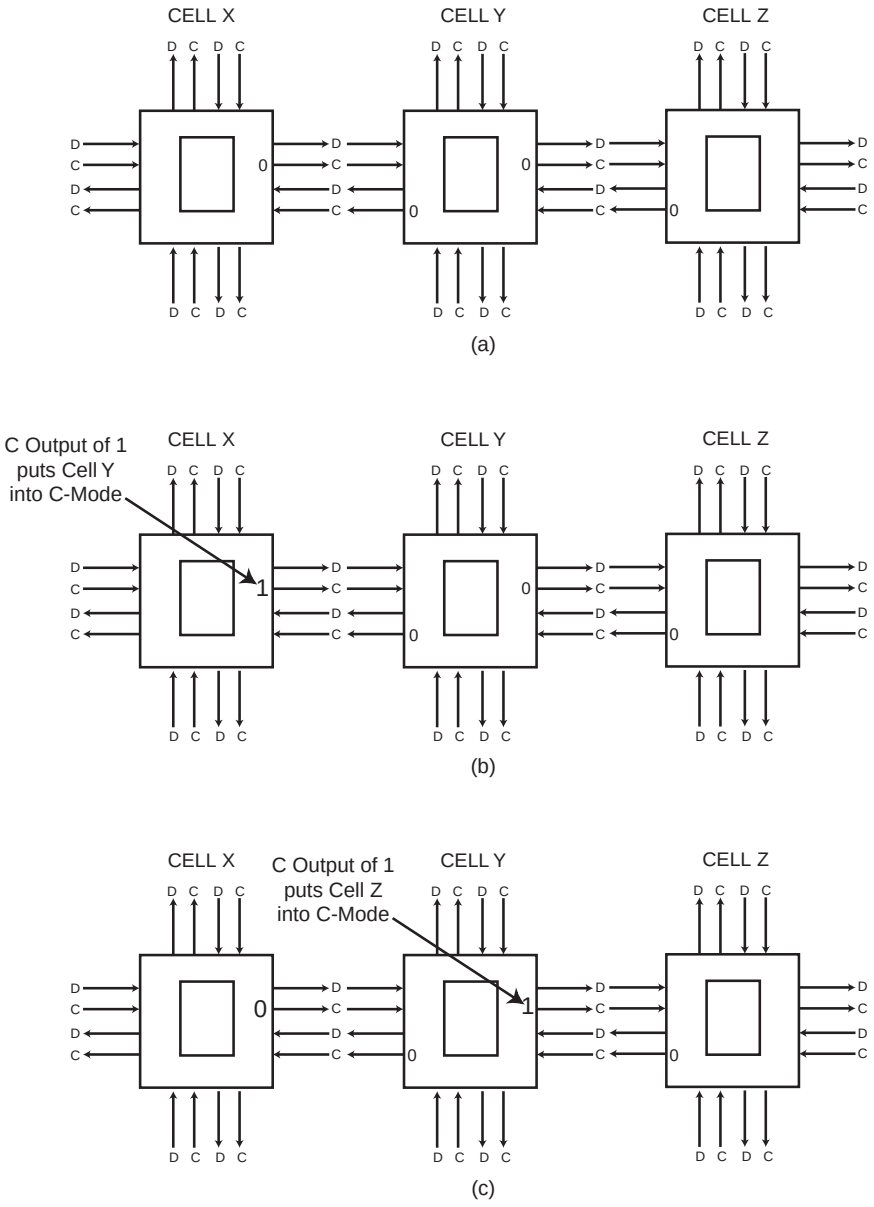

Fig. 1.8. Interaction of Cells' Modes. In (a), all three cells are in D Mode: each cell is reading inputs and producing outputs based on its current truth table contents and its D inputs. In (b), an input change (not shown) in cell X's D inputs has caused cell X to assert its C output to cell Y. Cell X has thus placed cell Y into C mode, and cell Y's truth table is now being configured. In (c), cell X is again outputting a 0 on its C output to cell Y. Cell Y has thus been returned to D mode. Based on cell Y's new truth table, cell Y is now asserting its C output to cell Z, and has thus placed cell Z into C mode. Cell Z is now being configured by cell Y.

In Figure 1.8.c, cell X has returned its C output to 0, and thus Cell Y returns to D mode. Cell Y is thus asserting its outputs based on the (new) contents of its truth table. In this example, the truth table indicates that Cell Y's C output to Cell Z is to be set to 1. This places Cell Z into C-mode, and Cell Y is now configuring Cell Z.

This example illustrates a very typical case: Cell Y was previously configured by a neighbor, but it is now itself configuring another neighbor. Within the Cell Matrix, there is a perfect interchangeability between subjects and objects of configuration operations. This is the essence of self-configuration and self-modification within the Cell Matrix.

1.2.5 Implications

There are a number of immediate implications arising from the architecture described above. The Cell Matrix **architecture** is infinitely scalable. Because only power and a single clock line are distributed throughout the matrix, there is no architectural impediment to scaling a matrix to whatever size is desired. Put another way (and, again, assuming a fixed dimensionality and interconnection topology), all sub-matrices of a given size are identical to each other, no matter what matrix they are embedded in: the structure of the cells and their interconnections is independent of the larger matrix to which they belong.

This means that two matrices can be combined into a large matrix simply by connecting the matrices to each other along an edge, i.e., connecting one matrix's edge cells' input to the other's edge cells' outputs, and vice versa. The architecture scales up without change (though of course the maximum possible latency increases). This also has interesting manufacturing implications (see Section 1.6).

Because configuration of cells is essentially a local operation, there is no such thing as *runtime vs. configuration time* for the matrix at large, no need to discuss *run-time reconfiguration*: the matrix is *always* running, and part of its running operation may include reconfiguration operations. Moreover, *partial configuration* (Schmit 1997) is the only type of configuration ever performed, since any single configuration operation affects only the neighbors of the cell being configured.

The Cell Matrix is completely homogeneous in structure. Cells are differentiated by their configuration information (truth table contents), but, at the underlying hardware level, all cells are identical to each other, just as are their interconnections to other cells. This has tremendously beneficial manufacturing implications. It also has positive implications for circuit reliability, since no piece of the matrix (or the circuits implemented on top of the matrix) is unique or irreplaceable.

Because of the capacity for self-modification in the Cell Matrix, high-level configuration mechanisms can be designed, tailored to the specifics of the target circuit, and then constructed out of cells. Moreover, the construction of the configuration mechanism *can itself be constructed* by using a previously-created configuration mechanism. In this way, configuring the Cell Matrix may closely resemble a traditional *bootstrap* process, wherein a simple circuit is first built using the limited control available from the edge of the matrix. This simple circuit is then used to configure a more complex circuit, which is then used to configure a more complex circuit, and so on, building

more and more complex circuits until the desired configuration has been achieved. Also, note that while a single set of commands may be used repeatedly to configure multiple Cell Matrix regions, it is still possible to introduce *differentiation*, including randomness, into the configured circuits.

Finally, because cells are configured by neighboring cells, it is possible for multiple cells to be configured simultaneously, either with the same configuration as each other, or with completely different configurations.

1.2.6 Status

The Cell Matrix architecture has been fully documented (Cell Matrix Corporation 2006a,b; Macias et al. 1999; Durbeck and Macias 2001c; Macias and Raju 2001). A variety of simulators and debuggers have been developed, as have various tools for developing circuitry on the matrix. Prototype tools for converting from abstract netlists to Cell Matrix configuration information have been developed (Macias 2006).

A number of fairly traditional circuits have been implemented on top of the Cell Matrix, including state machines, arithmetic units, memories, floating point processors, and cellular automata simulators. Section 1.4 will describe some of the less-traditional circuits that have been implemented, including circuits that utilize self-configuration.

Also, while the high-level behavior of the Cell Matrix is well-defined, there are multiple possible implementations of the Cell Matrix. For example, the original implementation (Macias et al. 1999) utilized a shift register for each cell's truth table. While this simplifies the design of each cell, it means that the entire truth table changes during a configuration operation. Later work produced a slightly more complicated cell implementation that utilizes a non-shifting memory that takes up a much smaller fabrication area (Durbeck and Macias 2001c). A further-modified cell incorporates bypass logic, to detect when a cell is acting as a wire and directly connect an input to an output, greatly improving signal transmission rates when cells are used as wires (Macias and Raju 2001).

1.3 Building Blocks of Self-Configuring Circuitry

This section will describe some of the *primitives* of self-configuration for the Cell Matrix, or the basic building blocks and techniques related to the implementation of self-configuring circuitry on the Cell Matrix. Section 1.4 will describe higher-level circuitry constructed from these blocks.

1.3.1 Cell-replication

Figure 1.9 shows a simple cell-replication circuit that copies the truth table of the source cell into the target cell. The cell in the middle is the *controller* of the configuration operation. The cell to be replicated (called the *source cell*) is on the right. The *target cell*, which will become a copy of the source cell, is on the left. When a 1 is sent

into the Northern D input of the controller, it asserts its C outputs on the left and right, thus placing the source and target cells into C mode. Each cell then begins outputting current truth table bits on one of its D outputs (on the side where C_{in} is asserted), as well as receiving new truth table bits on the same side's D input. The controller reads bits from the source cell, and sends them back into the source cell, thus rewriting the source cell's truth table while it is being read. Additionally, the controller sends the source cell's truth table bits into the target cell's D input, thus configuring the target cell's truth table as an exact copy of the source cell. After a sufficient number of clock ticks of the system clock (128 for four-sided cells, i.e., enough ticks to sample and write each of the truth table bits for a 16×8 truth table), the target cell's truth table will match the source cell's, and thus the target cell will behave exactly the same as the source cell: the source cell has effectively been replicated by the controller.

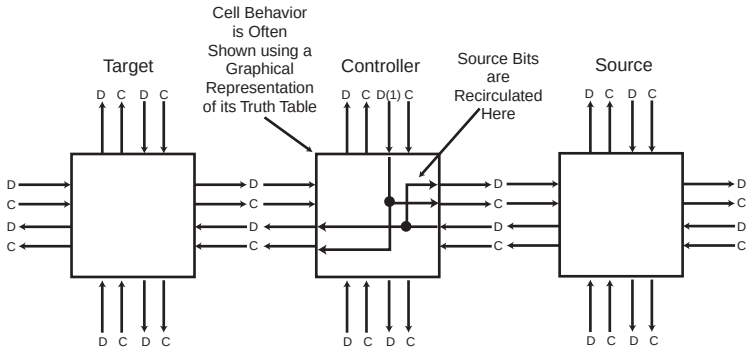

Fig. 1.9. Single-Cell Replicator. When a 1 is sent into the Controller's Northern D input, it places and Source and Target Cells into C mode, reads truth table bits from the Source, copies them back to the Source, and also copies them to the Target. After 128 ticks of the system clock, the Target will be an exact copy of the source. Because the Source Cell's truth table bits are loaded back into the Source Cell's D input, the Source Cell's truth table is left unchanged by this circuit. This is thus a non-destructive read.

1.3.2 Remote Cell Replication

Figure 1.10 shows a circuit that is similar to Figure 1.9, except that the source and target cells are not adjacent to the controller. Instead, the source and target are now some distance away from the controller, and cells located in between them are used to transmit C and D information between the controller and the source and target cells. The controller works the same as in Figure 1.9, except that it cannot directly control the mode (C or D) of the source and target cells. This makes the circuit in Figure

1.10 somewhat limited in its usefulness. A more useful approach involves the use of *multi-channel wires*.

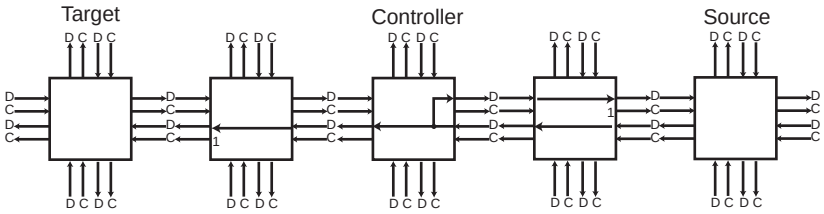

Fig. 1.10. Remote Cell Replicator. The Source and Target cells are in C-mode. The Source cell's truth table bits are read by the Controller, sent back to the Source cell, and also copied to the Target cell. Note that the Controller no longer directly controls the mode of the Source and Target cells.

1.3.3 Multi-Channel Wires

To control a cell, it is generally necessary and sufficient to control the C input, D input and D output on one of the cell's sides. The circuit shown in Figure 1.11 is an example of a structure for controlling non-adjacent cells, by utilizing two lines of intervening cells. Such a structure is called a *multi-channel wire*. In this circuit, the controller sends information along two lines of cells (each called a *channel*).

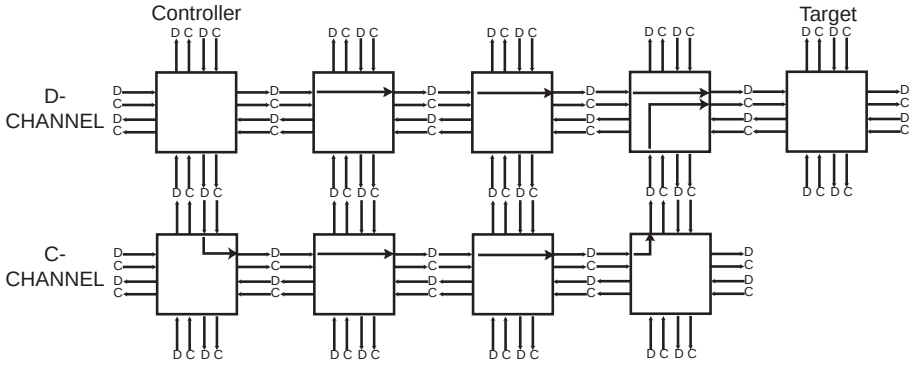

Fig. 1.11. Simple Multi-Channel Wire. The Controller sends new truth table bits into the Target via its own Eastern data output. Additionally, the Controller can now control the mode of the Target via its own Southern data output. This is a significant improvement over the circuit shown in Figure 1.10.

The bottom channel (called the "C Channel") controls the C input on the source and target cells, while the top channel (called the "D Channel") access the D input and

output on the source and target cells. Note that these lines are logical wires, or soft wires, rather than hard, physical wires: they are created by setting the truth tables of the cells to pass their input directly to their output. This primitive gives the controller more or less complete control over the source and target cells: the ability to place them in C mode, read and write their truth tables, and then return them to D mode.

There are other types of multi-channel wires, but they all have the same basic characteristic: they allow a set of cells to interact with one or more non-adjacent cells. By using the right types of multi-channel wires, a set of controller cells can thus configure cells that are not adjacent and not directly connected to itself.

While it is evident from this example that wires allow access to non-adjacent cells, it would appear that they only allow access to the cell adjacent to the end of the wire. This is not the case, however. If the target cell is treated as *itself* being a controller, then it is possible to access cells that are near, but not adjacent to the end of the wire. For example, if the target cell (call it *X*) is configured as shown in Figure 1.3.3, then data subsequently transmitted to cell *X* will, in fact, be used to configure the cell *below* cell *X*: the cell shown in Figure 12 effectively moves the location of the wire's target cell.

Therefore, even though a wire can directly control only the cell adjacent to its end, it can *indirectly* control non-adjacent cells using intermediate cells such as that shown in Figure 12.

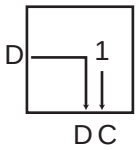

Fig. 1.12. A target cell which can be used to configure a cell *near* a wire's target cell.

Repeated application of this technique could, in theory, be used to gain control over a cell located *anywhere* within the matrix. However, this technique is limited in its usefulness, since accessing cells "n" locations away requires on the order of 2^n steps. Thus, wires' only practical use during configuration is to manipulate cells *near* their end. This would be a severe limitation of the Cell Matrix's nearest neighbor topology, if not for the concept of *wire building*.

1.3.4 Wire Building

Special kinds of wire building permit a cell to access any cells within a Cell Matrix. While wires can only be used to control cells adjacent to their ends, **those wires themselves are built out of cells**. In fact, it is possible to design a wire that allows cells at its end to be configured in order to make a new piece of the wire, i.e., to extend the wire. Beginning with a short wire, cells at its end can be configured to make the wire one cell longer. That longer wire can be used to configure the cells at the new end, thus making the wire another cell longer, and so on. This process can be repeated

indefinitely (as long as there are cells available), thus allowing a set of controller cells to access cells arbitrarily far away. Moreover, it is possible to create wires that have turns, and again these turns can be created by the controller. This means a set of cells can, in fact, access any cells within the matrix, through the use of proper wire-building techniques. Note that this permits remote control of cells, even though the underlying primitives used are all strictly neighbor to neighbor. Remote control permits external control of the system, but it also permits the smallest unit of a complex system to be multi-celled to an arbitrary size, which is convenient for most applications, including most work in self-organizing systems. In Section 1.4 we describe an application that uses a Supercell as its unit, which contains 270×270 cells.

Figures 1.13 shows a sample two-channel wire that is *extendible*. As in the wire of Figure 1.11, the upper channel transmits a D signal, and the lower channel transmits a C signal. The cell labeled "*" is again called the target cell, and as in Figure 1.3.3, both Cell (*) and Cell (**) can be easily configured. However, unlike the two-channel wire shown in Figure 1.11, **all of the D channel cells are identical**, and **all of the C channel cells are identical**. This is accomplished through the use of a feedback signal: each cell within the D channel asserts a 1 to its south, which the corresponding C channel cell transmits to the previous cell of the C channel. Therefore, the end of the wire is identified not by a differently-configured cell, but rather by the lack of this feedback signal. Thus, by simply creating a new pair of D Channel and C Channel cells at the end of the wire, the wire is effectively extended, i.e., the location of the target cell is shifted one cell to the right.

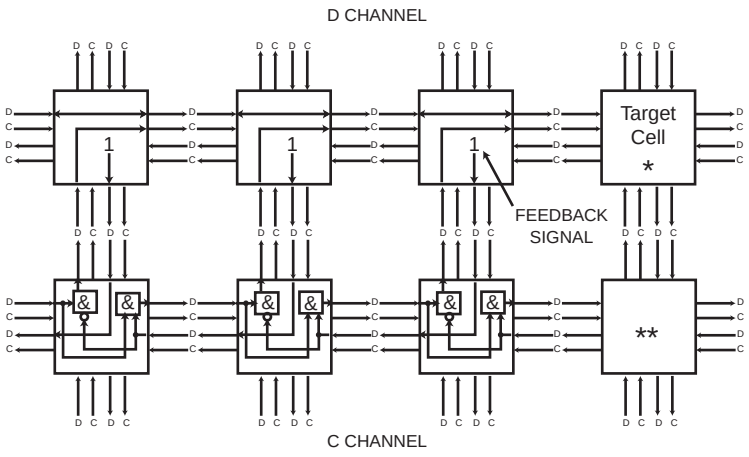

Fig. 1.13. Two Channel Extendible Wire. The D Channel transmits configuration information for the Target Cell. The C Channel controls the mode of the Target Cell. The Feedback Signal is used for autonomous determination of the location of the wire's end. If Cells (*) and (**) are configured as new channel pieces, then the wire will automatically be extended.

This is the essence of wire building. While different sequences are needed for different types of extensions (such as turns) and for different types of wires (such as 3

channel wires), the basic mechanism is the same as in Figure 1.13. These mechanisms are described in more detail elsewhere (Macias 2001).

Multi-channel wires and associated wire-building techniques can be used to access cells anywhere within the matrix. This raises the question, "What does one do with such access?" There are many answers to this, and in the remainder of this section, a few general examples will be presented. Section 1.4 will discuss more-specific examples.

1.3.5 Cell Testing

Given access to the C input and D inputs and outputs of a target cell, it is possible to perform a variety of tests on the target cell, to ascertain its health, i.e., to determine if it is operating as expected. For example, the cell's truth table can be loaded with 0's, and then the D output examined while the D input is toggled between 0 and 1. This would detect shorts between input and output, as well as detecting stuck-at-one faults inside the truth table memory, or along the D input or D output paths. A second example is that a set of certain bit patterns can be loaded into the cell, and then read back out and compared to the loaded pattern: different alternating bit patterns can be used to detect shorts within the truth table memory, based on the physical layout of the memory within the cell. This fault testing work was developed and successfully conducted on defective hardware for the Cell Matrix architecture using the above-described multi-channel wire building to reach each cell (Durbeck and Macias 2002).

1.3.6 Circuit Building

The question of how to *bootstrap* a Cell Matrix remains, that is, with no direct access to the vast majority of cells within the Cell Matrix, how can a Matrix be populated with the desired set of truth tables, particularly given that the Matrix is always running, and thus, truth tables are in use from the moment they are in place. However, all the necessary building blocks have already been presented. Figures 1.14.a-1.14.f show a sample bootstrap sequence. Note that this is not the only possible way to bootstrap a region of the Cell Matrix. It is a pedagogically interesting example because it is one of the simplest and most straightforward, but it is not used in typical practice, because it is one of the slowest ways to configure a region of cells.

In Figure 1.14.a, a two-channel wire is built from West to East, extending just one column of cells shy of the Easternmost corner of the region of interest. The target cell in the corner of the region is then configured.

In Figure 1.14.b, the wire has made a corner, and is extended one step to the South. This extended wire is then used to configure the next target cell (*).

The wire is then extended South another step, and a third target cell is configured to the East, as shown in Figure 1.14.c. This process continues, until, as in Figure 1.14.d, an entire column of target cells has been configured, along the Easternmost edge of the region of interest.

In Figure 1.14.e, the wire has been *broken*, i.e., the end of the wire is returned to the original entry location into the region of interest. The wire is again extended to the East, but stops one cell earlier.

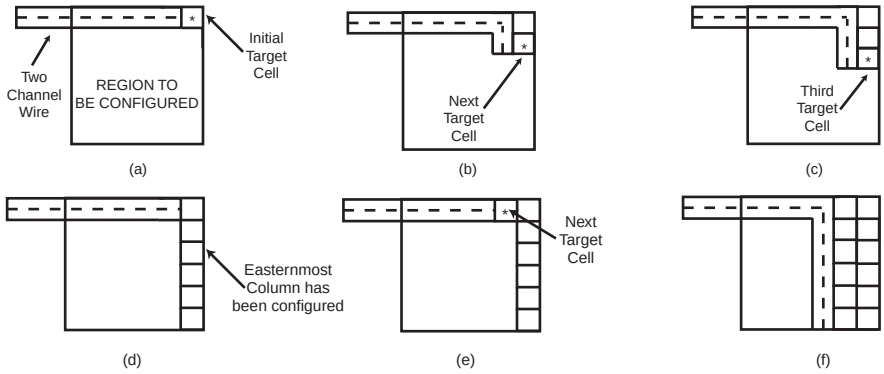

Fig. 1.14. Configuration of a Region. In (a), a two-channel wire is built into the region of interest, and is used to configure cell (*). In (b), the wire has been extended with a corner, and the next target cell is configured. In (c), the wire is extended further to the South, and a third target cell is configured. In (d), the Easternmost column has been completed. (e), shows the beginning of the second column's configuration: the wire has been broken and re-built, but ends one cell shy of the previous extension. In (f), the second column has been completely configured. This process is repeated until the entire region has been configured.

The wire then turns a corner, and the above steps (configure/extend) are repeated, configuring a second column of cells, as shown in Figure 1.14.f.

The above steps are repeated, until the entire region of interest has been configured. Note that this technique cannot be used exactly as described for configuring the Westernmost columns, since the wires themselves have a width to them. The easiest way to address this is to avoid this edge case by imagining that the region of interest as being one wire width wider than it really is, and leaving the Westernmost columns (which are not actually of interest) unconfigured.

There are numerous enhancements to this basic scheme, including techniques to avoid completely rebuilding the West-to-East wire after each column pass. Parallel configuration is also feasible, and will be discussed briefly in Section 1.4. Also, note that the configuration of cells that assert their C outputs requires special consideration, since such outputs could interfere with the configuration of the wires that are being used to configure the region's cells.

1.3.7 Circuit Reading

Using circuits and sequences similar to the bootstrap method described above, it is possible to non-destructively read a set of cell configurations from a region of the matrix. The technique is similar to bootstrapping, but utilizes a *reversible* wire, i.e., one that can not only be extended a single step, but can also be *shortened* a single step. The basic technique is shown in Figures 1.15.a-1.15.f. For simplicity, this illustrates the reading of a single (one-dimensional) line of cells only. Note that implementation of a reversible wire requires three channels.

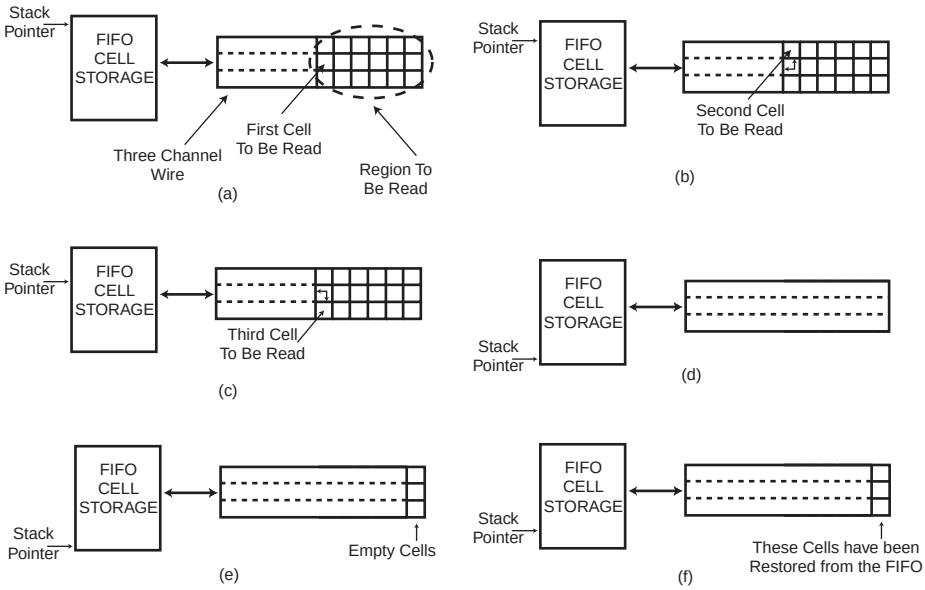

Fig. 1.15. Non-Destructive Read of a Region of Cells. In (a), a three-channel wire has been built to the edge of a region to be read. A first cell is read, and its configuration is stored in the FIFO. In (b), that first cell is used to read a second cell, which is also stored in the FIFO. In (c), a third cell is read and stored, after which the wire will be extended one step. In (d), the entire region has been read, stored and overwritten with the wire itself. In (e), the wire is reversed (backed up one step), and in (g) the Easternmost cells have been restored form the FIFO.

In Figure 1.15.a, the wire has been built to the East, to the beginning of a set of cells whose contents are to be read. The cell directly ahead of the wire (i.e., to the East of the D channel) is read, and its truth table configuration is stored in a temporary repository (a FIFO). That cell is then configured to allow reading of the cells to the North and South of it, with those cells' configurations also being stored in the temporary repository. This is shown in Figures 1.15.b and 1.15.c, respectively.

The three-channel wire is then extended, and the process repeated. In Figure 1.15.d, the wire has extended all the way to the East. Again, edge cases need to be considered, but can be neglected by imagining the region of interest to be larger than it actually is. At this point, all of the cells occupied by the wire have been reconfigured from their initial configuration (in order to implement the extended wire), but their initial configurations have been read (and presumably processed by some circuitry outside that shown in these figures). Also, **those configurations have been stored in a temporary storage location** (which can be as simple as a set of cells arranged in a two-way shift register, i.e., a FIFO).

In Figure 1.15.e, the wire is reversed a single step (using the third channel), and in Figure 1.15.f, the previously-stored configurations are restored to the cells near the end of the wire. These two steps are repeated, until the entire row has been restored.

For a two-dimensional region, another pass would be made to the south of the original West-East wire, thus reading the next three rows of cells. At the conclusion of this, the cells within some region of interest will all have their initial configurations, but a copy of those configurations will have been sent by this circuit to some other circuitry that will perform analysis, make a new copy, vote on truth table contents among multiple copies, or conduct some other function.

Note, however, that while the above technique will read the configuration of cells without (permanently) changing them, it does **not** read the state of cells, i.e., the values of their inputs and outputs, and similarly does not preserve their state. State reading and preservation would require additional circuitry built into the circuit itself, since changing the configuration of a single cell can, in general, alter the state of the entire circuit.

These are a few detailed examples of techniques related to self-configuring circuitry. They form a base of primitives or building blocks that are composed to create more complex functions and circuits. The next section will describe larger-scale applications of these techniques to the implementation of circuits that exhibit distributed management and control.

1.4 Distributed Management and Control in the Cell Matrix

There are a number of examples of how the Cell Matrix can be used to manage various tasks related to its own operation and maintenance. While non-Cell Matrix systems could be designed specifically to implement any of these examples, the advantage of the Cell Matrix architecture is that *it supports* **all** *of them*: the Cell Matrix architecture does not have to be modified in any way in order to implement these systems.

1.4.1 Hardware Error Checking

The wire building and cell testing techniques described above can be used to test individual cells within the matrix, to ascertain their proper functioning. Moreover, since all that is required to perform these tests are basic logic circuits and simple state machines, these tests can be performed by circuitry within the matrix itself. This offers a number of interesting opportunities.

For example, once a small initial set of cells is known (say, via conventional validation techniques) to be functioning properly, and a state machine is built to perform subsequent cell tests, cells can be tested, verified, and then used to build longer wires, allowing testing of more-remote cells. Other than the initialization issue, this eliminates the question of “what if the test circuit itself is defective?” since only known-good cells will be used in extending the test circuitry (Macias and Durbeck 2004; Durbeck and Macias 2002; Macias and Durbeck 2002).

By running multiple test circuits, cross-checking can be performed among multiple testers. Basic N-way redundancy could be used to verify the initial circuitry, after which a single copy would suffice (as far as manufacturing defects are concerned: transient errors are a different consideration).

This approach also allows parallel testing to be performed. Again, starting from a single test circuit, multiple testers can be configured from known-good cells, and these testers can operate in parallel to test multiple regions simultaneously, and then construct more parallel testers. This can reduce test time to $O(n^{1/2})$ for n cells in a two-dimensional matrix, and $O(n^{1/3})$ in a three-dimensional one (Durbeck and Macias 2001d; Macias and Durbeck 2002, 2004).

Test results can be stored in something similar to a "bad block" list (Duncan 1989), and this list used in subsequent configuration operations. If a place-and-route algorithm were implemented directly on the Cell Matrix hardware, it would simply note these defective cells as being unavailable for placement or routing, and would thereby avoid them in creating compiled circuits.

For handling run-time defects such as single event burnout (Waskiewicz et al. 1986) or single event gate rupture (Fischer 1987), these tests could be performed periodically. Multiple copies of circuitry can be maintained, with copies taken offline individually, their underlying cells re-tested, and their configuration adjusted as needed to avoid newly-defective cells.

1.4.2 Autonomous Fault Handling Through Autonomous Circuit Building

We have devised a methodology for implementing a desired target circuit on top of the Cell Matrix in a way that allows that system to configure itself in order to avoid defective regions of the matrix (Macias and Durbeck 2004, 2002; Durbeck and Macias 2001a,b). If new defective regions are later found or suspected to be present (for example, because some sort of built-in self test has failed), the system can be given a single "REBUILD" command, and it will locate, isolate and avoid all defective regions, while re-implementing itself using only good cells. The goal was to have these operations performed by the system itself, with a minimal amount of external intervention required. This work combines the above techniques of hardware error checking with some bio-inspired concepts in self-organization (Mange et al. 2000).

This approach utilizes the concept of a *Supercell*. This is a general term for a collection of contiguous Cell Matrix cells configured to perform a variety of functions while still retaining the underlying self-configurability of individual cells. In our self-repairing circuit building work (Macias and Durbeck 2002, 2004) the Supercell first performs a number of *initialization* functions, including:

- testing a region of the matrix for defective Cell Matrix cells;
- configuration of new Supercells on known-good regions;
- activation of isolation circuitry within good Supercells, in order to prevent any interference from bad Supercells; and
- sharing of configuration information among a network of Supercells in order to configure new Supercells in multiple regions in parallel.

The purpose of the initialization stage is to tile a region of the Cell Matrix with known-good Supercells, while isolating defective cells. Note that this part of the system's operation requires a set of configuration strings to be sent into the empty matrix. These configuration strings depend on the high-level circuit to be implemented, but are

completely independent of the location of any defects in the Matrix (since the location of such defects is assumed to be unknown). All subsequent steps are performed by the collection of Supercells themselves, without any further external intervention.

Following initialization, the system enters a *differentiation* phase. In this phase, Supercells assign themselves unique integer IDs, so that they can be differentiated from each other. Without such an assignment, all Supercells are identical to each other. This assignment is accomplished through the collective operation of the entire set of Supercells. Differentiation changes the contents of two ID registers contained within each Supercell:

- one ID contains a position-dependent integer, which is simple to assign (by incrementing an incoming neighbor's ID and passing that to other neighbors), but the set of assigned integers is not necessarily contiguous; and
- a second ID that is position-independent, and whose collection is guaranteed to form a set of contiguous integers.

When the initial configuration strings are developed, an abstract representation (called the "genome") of the final target circuit is coded inside the strings. The genome is simply a netlist, specifying the components of the final circuit, along with their input-to-output interconnections. What is **not** specified is the particular locations of those components in the matrix, nor the paths that will be used for their interconnections (since doing so would require knowledge of the locations of defective Cell Matrix cells).

After unique IDs have been assigned, each Supercell compares its ID with the ID of components stored in the final circuit's genome (this is why contiguous IDs are required). Using the information inside the genome, each Supercell thus knows which component it is to implement in the final circuit. Each Supercell then configures inside itself its piece of the final circuit.

Following differentiation, the collection of Supercells must be wired together in order to implement the final target circuit. This requires first determining pathways from component to component, and then creating communication channels along those pathways. Both of these steps are performed by the Supercells themselves, without any external intervention. Path-finding is done using a greedy algorithm, which picks the shortest path from component to component. Channel creation is performed by utilizing pre-existing pieces of channels inside each Supercell, and by configuring cells near the junction of these channel pieces, in order to form continuous pathways. Note that some of these channel pieces cross within the Supercell, to allow the creation of crossed communication pathways.

The final Supercell design consisted of 270×270 Cell Matrix cells. This was intended only as a proof-of-concept, and represents neither the smallest nor most-efficient Supercell for the given problem. There are ways to perform more traditional fault tolerance for the Cell Matrix architecture that have been developed and analyzed (Saha et al. 2004; Macias and Durbeck 2005a), but the point of the Supercell work was to demonstrate autonomous, self-organizing circuitry, and faulty hardware was simply the impetus to which the system responded in order to modify its behavior.

1.4.3 Self-Replication

Figures 1.16.a-1.16.e show the operation of an entirely self-replicating circuit on the Cell Matrix. The circuit is comprised of two main parts. The upper-left region of the circuit is called the "Main Grid," and is a state machine that generates bit sequences and sends them into three wires. The right half of the circuit is a copy of the left half, but with additional space (two rows of empty cells) between each row of non-empty cells. The right-hand circuit is called the "Exploded Grid," since it is a copy of the Main Grid on the left, but with the rows spaced out vertically.

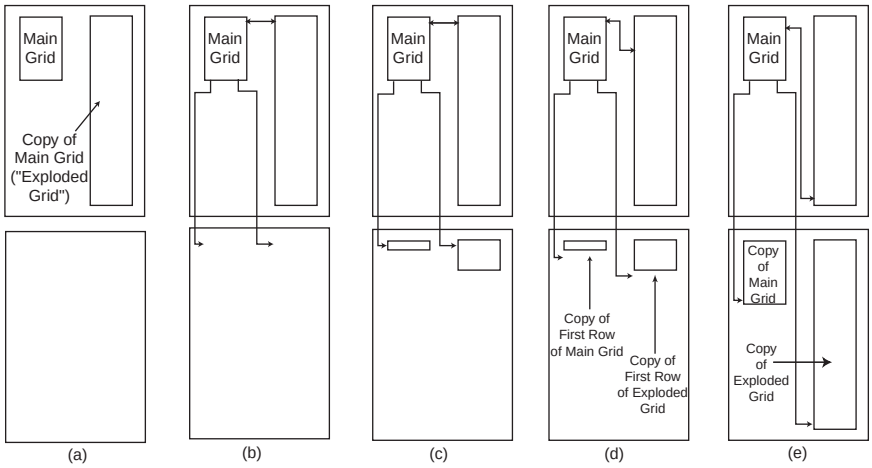

Fig. 1.16. Self-Replicating Circuit. (a) shows the initial configuration. The Main Grid is a circuit that will read the Exploded Grid, and produce a copy of it and itself. In (b), three wires have been built, extending into the Exploded Grid and the initially-empty regions to the South. In (c), the first row of the Exploded Grid has been read and used to reproduce the first row of the Main Grid and the Exploded Grid in the region to the South. In (d), the three wires are moved in preparation for reading the next row of the circuit. In (e), all rows have been read and copied, creating an exact copy of the original circuit in the region to the South.

The reason for using an Exploded Grid is that we do not want the circuit to actually be operating: it is intended to supply a *data* version of the circuit being replicated. Since the circuit itself is intended to modify other cells, we want to carefully control the behavior of this copy. So the cells in the Exploded Grid that can assert their C outputs are kept permanently in C mode, so that their C outputs are constantly forced to 0. This is achieved by pairing such cells with other cells, called "Guard Cells."

The bitstreams generated by the Main Grid extend the three wires as follows:

- one wire is extended into the Exploded Grid, in order to read the cells within the first row of that grid;
- another wire is extended into an empty region below the Main Grid, and will create a copy of the Exploded Grid there, but without any inter-row spaces; and

- the third wire is extended into an empty region below the Exploded Grid, and will create an exact copy of the Exploded Grid.

Figure 1.16.a shows the initial circuit. Figure 1.16.b shows the circuit with the three wires immediately prior to reading the first cell from the Exploded Grid. In Figure 1.16.c, the first row of the Exploded Grid has been completely read, and two copies of it have been made in the region below the original circuit. A single row has now been configured in the new Main Grid, and three rows have been configured in the new Exploded Grid (one row of Main Grid circuitry, and two rows of Guard Cell circuitry).This process is somewhat analogous to the translation and transcription steps found in the replication of DNA (Arms and Camp 1987).

Figure 1.16.d shows the circuit once the wires have been adjusted for reading the second row of the Exploded Grid. Of course, the wires extending into the original and new Exploded Grids require more extension steps than the wire in the new Main Grid.

The above steps are repeated for each row of the Exploded Grid. Figure 1.16.e shows the final state of the system, where an exact copy of the original circuit has been made to the South. Typically, the lower-rightmost cell would be configured to output a "GO" signal into the circuit, which would trigger the execution of the above circuitry. Thus, as soon as a new copy of the circuit is created, it immediately begins making a new copy of itself.

This is, in itself, not necessarily useful, but is a useful building block to which a number of various enhancements can be added. For example, after being placed on a Cell Matrix, the circuit could make a copy of itself to the South. That copy could make a copy of *itself* to the South, and that copy could make a copy of itself, and so on, thus creating a single column of copies of this circuit.

The height of the column could be hardwired into the circuit, for example, by incrementing a counter within each circuit, and only replicating a fixed number of times. Alternatively, the height could be determined dynamically by the presence of a marker in the matrix indicating the desired extent, or the circuit can itself determine when it has reached the Southern edge of the matrix (by noting that cell configuration operations no longer work). Once a leftmost column has been created, the bottom circuit can signal to the other circuits in that column to begin replicating to the East, resulting in the parallel building of a new column. Note that each circuit in the column replicates in parallel with every other circuit in that column. Thus, whereas creating the initial column (containing, say, n copies of the circuit) required n replication cycles, creating the second column requires **only one replication cycle**.

Generation of each subsequent column will also require the same time as a single replication cycle. To create an $n \times m$ array of these circuits would thus require n replication cycles to configure the first column, plus $m-1$ replication cycles to create each of the remaining $m-1$ columns, for a total of $n+m-1$ replication cycles. This is extremely better-than-linear performance. In fact, configuring n copies of this circuit requires on order of only $n^{1/2}$ steps. For a three-dimensional Cell Matrix, configuration time for n copies is a mere $n^{1/3}$ steps. This is an extremely efficient way to configure large regions of a Matrix.

Of course, we are usually interested in something more than simply filling the matrix with copies of a single circuit. The self-replicating circuit is intended to be a *carrier* of additional circuitry.

1.4.4 Fully Autonomous Self-Configuration

The above self-replicating circuit can be used to make copies of the Supercells described previously, which will then autonomously implement a desired target circuit. Collectively, this represents a fully autonomous, fault-handling, self-configuring system. This circuit can be thought of as a seed, or perhaps a biological cell. Upon placing a single copy of it inside an empty matrix, it begins to replicate, filling a region of the matrix with copies of itself. Once a sufficient number of copies have been created, they begin to differentiate and specialize, and then work together to implement some higher-order function. Moreover, this is done in a dynamic manner, with the exact configuration of the final circuit dependent on the environment (specifically, the location of defective cells within the matrix).

1.4.5 Hardware Compilation

Using the techniques described above, it is possible to perform compilation of algorithms not into software, but directly into hardware. This is already an active research area of reconfigurable logic (Page 1996). However, with the Cell Matrix as the underlying hardware substrate, it is possible to design systems that are *self-compiling*, i.e., the circuitry that produces the final compiled circuit can itself be running on the Cell Matrix.

One area in which such a setup would be useful is in implementing a Just-In-Time (JIT) (Deutsch and Schiffman 1984) compilation system. There is a huge parameter space within which algorithms could be developed. For example, the wordsize of an arithmetic unit can be adjusted based on the characteristics of data being processed. This might change over time, and circuit characteristics adjusted accordingly. Similarly, the number of registers available in a general-purpose processor (implemented on the Cell Matrix), or the character size of a hardware string processor could be adjusted over time. Sequences of operations that occur repeatedly could be analyzed, and new hardware synthesized to implement their collective function directly in hardware. Such hardware could be dismantled, and the underlying cells re-used, if the circuitry is not utilized for some period of time.

1.4.6 Hardware Operating Systems

Having self-configurable hardware, it is possible to consider hardware analogs of various software concepts, especially concepts related to operating systems. This leads to the concept of a *hardware operating system*.

For example, combining wire building techniques and bootstrap mechanisms, one can easily imagine the notion of a *hardware library*, wherein circuits consisting of Cell

Matrix cells are stored, available for retrieval and replication elsewhere in the matrix, just as software libraries store code that is re-used in other programs.

As another example, the notion of virtual memory could be extended to *virtual hardware*, where a matrix appears to have more hardware than actually exists. By storing cell configuration information (and utilizing appropriate compression mechanisms), and using it to configure cells as needed, it is possible to design a system on the Cell Matrix that emulates a matrix that is larger than the physical matrix on which it resides. Such a system would, effectively, intercept accesses to non-existent cells, create them on-the-fly, and redirect requests to those newly-created cells. These cells would be located virtually in a fixed location, but *physically* might be located somewhere different.

Closely related to this notion of virtual hardware is *hardware swapping* and *hardware timesharing*, where a single Cell Matrix is shared by multiple applications, which are loaded and unloaded from the matrix's cells, so that a single set of cells are used for more than one application. Such a system would probably employ a double-buffering mechanism, so that while one circuit is executing on one region of the matrix, another region would be configured with the second circuit to be executed. Once that circuit is ready, and the desired time slice has expired, that second circuit would begin executing, while the cells of the first circuit would be re-configured to implement the third circuit, which would eventually be allowed to run while the fourth circuit was implemented, and so on. Once the last circuit was given a time slice, the first circuit would again be configured and allowed to run. Other than the need to double-buffer (since configuring a circuit takes a non-negligible amount of time), this is highly analogous to swapping and timesharing of software. Again, as described above, consideration must be given to reading, preserving and restoring not only the configuration of each cell within a running circuit, but also the state of each cell, meaning its outputs and inputs.

1.5 Extension to the Analog Domain: The Songline Processor

"...the labyrinth of invisible pathways which meander all over Australia and are known to Europeans as 'Dreaming-tracks' or 'Songlines'; to the Aboriginals as the 'Footprints of the Ancestors' or the 'Way of the Law.' Aboriginal Creation myths tell of the legendary totemic being who wandered over the continent in the Dreamtime, singing out the name of everything that crossed their path - birds, animals, plants, rocks, waterholes - and so singing the world into existence" (Chatwin 1986).

The previous descriptions of a self-configurable processor are all rooted in the digital domain: one where inputs and outputs are binary in nature, i.e., where each input has one of only two possible values. This can be extended by allowing inputs and outputs to have values within a continuous range. For example, instead of standard TTL-level signals, inputs and outputs can be allowed to have values anywhere between 0 and 1 volts. Such an extension not only changes the domain and range of the function, but also the basic characteristics of the mapping function. In particular, a truth table with discrete rows will no longer suffice for describing a cell's input-to-output mapping.

As will be seen in what follows, extension of the Cell Matrix architecture in this way leads to a very different type of processor whose inputs, outputs and programs are, in some sense, akin to music: time-varying signals that are copied from one element to another in C-mode by playing and recording them. In D-mode, the information stored within a song is extracted by sampling the song at a particular point in time. This is not entirely dissimilar to aspects of *Songlines*, which are passed from person to person through singing, hearing and memorizing, and whose information is extracted by singing/listening to the song at a particular point in time (corresponding to where one is geographically in their traversal of the Songline). For this reason (and with great respect), this extended version of the Cell Matrix is called a *Songline Processor*. It's internal mapping function can be called a "song," and a particular value derived from the mapping may be called a "note." For clarity in what follows though, we'll stick to the mathematical terminology of "function" and "value."

Consider a cell with a single input and a single output. For the binary version of a cell, a truth table consisting of a single entry - say an input x - will suffice. The output values corresponding to $x = 0$ and $x = 1$ must be specified, and this completely defines the characteristics of the cell. Such a truth table can be stored by simply recording two bits: basically $f(0)$ and $f(1)$, where $f(x)$ is the cell's mapping function that turns a single input bit into a single output bit.

But when the inputs and outputs are real-valued, the mapping function $f(x)$ is now a real-valued function of one real variable. To store a complete specification of this function in a truth table would require an infinite number of entries (one for each possible real-valued input value), with each entry specifying a single real-valued output. Table 1 shows an approximation of such a table for a sample function ($f = \sqrt{(}x), x\epsilon[0, 1]$):

x	$f(x)$
0.000	0.000
0.001	0.032
0.002	0.045
0.003	0.055
...	
0.998	0.999
0.999	0.999
1.000	1.000

Table 1
Approximation of function $f(x) = \sqrt{(}x)$

Of course, this is only an approximation of an exact truth table, which would require an infinite number of rows (with an infinitesimal change in x from row to row).

This is complicated further in the case of a two-input cell, whose mapping function $f(x, y)$ is a function of two real-valued inputs. Table 2, for example, shows an approximation to the function $f(x, y) = x \times y$.

x	y	$f(x,y)$
0.000	0.000	0.000
0.000	0.001	0.000
0.000	0.002	0.000
...		
0.000	0.999	0.000
0.000	1.000	0.000
0.001	0.000	0.000
0.001	0.001	0.000
0.001	0.002	0.000
...		
0.001	0.999	0.001
0.001	1.000	0.001
...		
0.999	0.000	0.000
0.999	0.001	0.001
0.999	0.002	0.002
...		
0.999	0.999	0.998
0.999	1.000	0.999
1.000	0.000	0.000
1.000	0.001	0.001
1.000	0.002	0.002
...		
1.000	0.999	0.999
1.000	1.000	1.000

Table 2
Approximation of function $f(x,y) = x \times y$

Clearly one can estimate a mapping of any number of input variables using such discretization of the input space. While this is not an ideal for physical implementation, it can be a useful model to keep in mind.

For working purposes, we consider our basic programmable cells to be four-sided, with (again) a C and D input and output on each side. These cells are arranged in a 2-D layout with each cell having four neighbors. We continue to designate the sides as N, S, W and E, but we now define a cell's "truth table" with a series of mathematical expressions describing each D and C output as a function of its four D inputs. Details related to storing such functions will be discussed below in the Implementation section. As will be seen, implementation difficulty increases significantly (for a variety of reasons) as the number of sides increases.

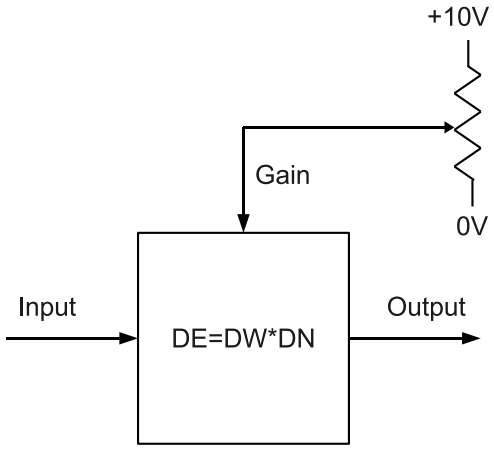

Fig. 1.17. Single-Cell Amplifier. The variable resistor sets the desired gain, which multiplies the input signal to produce an amplified output.

The tradeoff for this difficulty of implementation is a richly-powerful computing paradigm, where, for example, we can compute square roots with a single pre-programmed cell; or the product of two numbers with a differently-programmed cell. Moreover, the fact that these cells are operating on inherently-analog signals (as opposed to digitizing analog inputs and processing them discretely) has interesting implications application-wise. For example, Figure 1.17 shows a simple amplifier circuit, comprised of a single cell.

Here, the input DN controls the degree of amplification; DW is the input signal; and DEout is the amplified output. By adjusting the variable resistor to set DN anywhere from 0 to 10 volts, the input is amplified accordingly (up to a maximum output of 1V).

Figure 1.18 shows an implementation of a continuous-valued flip-flop. The incoming data signal is sent to the D input, and the GATE control is sent to the G input. The latched value can be read from output Q. The pair of cells operate together to create a feedback loop that traps a particular value between them. When G is high ($G > 0.5$ V in this case), the incoming signal is sent to the feedback cell, which re-sends it to the initial cell. As the input signal changes, its value is continuously updated in the feedback loop. But when the gate closes ($G < 0.5$ V), the incoming signal D is ignored, and the value received from the feedback cell is re-circulated back to that cell, creating a closed loop that traps a single value. This is effectively a sample-and-hold circuit.

Figure 1.19 shows a basic differentiation circuit. The incoming signal is sent into DW; this signal is passed to the cell on the right, which returns it to the cell on the left. The cell on the left computes DW - DE, and sends the difference to DSout. Thus, for an incoming signal $f(t)$, DSout=$f(t + \delta t) - f(t)$ where δt is the time it takes the incoming signal to enter and leave the cell on the right. Of course, a scalar multiplier

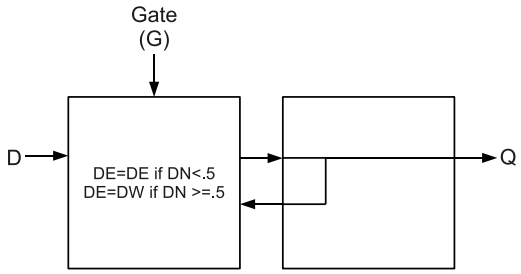

Fig. 1.18. Continuous-Valued Flip-Flop. Raising the Gate input above 0.5V allows input D to be loaded into the device. Dropping the gate below 0.5V traps the loaded value between the two cells. Q presents the output value.

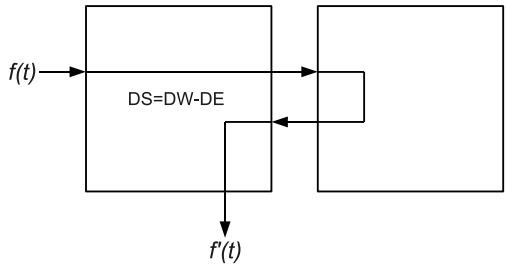

Fig. 1.19. Differentiation Circuit. $f(t)$ can be any time-varying input signal; its derivative with respect to time is produced from the bottom of the leftmost cell. The derivative is scaled by an amount related to the propagation delay of the cells.

can also be supplied to the cell on the left to adjust the output range based on how quickly the input signal is changing. Similar designs can be created for integrating (summing) an incoming signal; such designs also require only two cells.

Figure 1.20 shows a ramp generator. The rightmost cell is again just a feedback path. The cell on the left receives the fed-back signal, adds an increment to it (which is supplied by the DW input), and sends the sum to the right. The value of the Increment input (in conjunction with the input-to-output time of the cells) determines how quickly the output rises. For example, to achieve an output frequency of 1Hz, supposing the total propagation delay through the two cells is τ, we would need an increment value of 0.9τ.

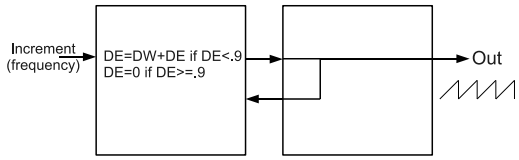

Fig. 1.20. Ramp Generator Circuit. The output rises to 90% of a cell's maximum output value and is then reset to 0. The rate of increase depends on the Increment input, which thus adjusts the frequency of the output.

The equations in the leftmost cell cause it to rollover back to 0 anytime its outputs exceeds 0.9V. In this setup, the output Q is thus a repeating sawtooth, rising from 0.0 to 0.9V and then returning to 0.0V. Of course, the cell on the right could also amplify the output signal to increase the maximum output to 1.0V. The frequency of this waveform depends on the value of the Increment input: the larger the increment, the more quickly the output rises, and thus the higher the frequency. Of course, if the input-to-output time is large compared to the period of the generated waveform, the output may be badly discretized. But assuming a small propagation delay, the output will rise relatively smoothly. Figure 1.20 is thus a voltage-controlled oscillator.

1.5.1 C-Mode

The above descriptions all assume previously-programmed cells, connected to build a static (non-changing) circuit. This is analogous to using digital/binary Cell Matrix cells to implement digital circuits that do not change. While certainly useful, this is only part of the Cell Matrix story. So is the case for a Songline Processor, which incorporates its own notion of a C/D-mode state.

Recall that for a binary Cell Matrix, the mode of a cell is also a binary variable associated with a cell's C input: If $C = 1$ then the cell is in C-mode; and if $C = 0$ the cell is in D-mode. These modes are fundamentally different from each other, and the cell completely changes from one mode to the other based on transitions of its C inputs.

In a Songline system, the *mode* of a cell is a real-valued variable (say between 0 and 1), and the corresponding behavior of the cell is a mixture of its pure D and pure C mode behaviors. The pure behaviors are as follows (the use of the terms "bit" and "truth table" will be clarified below):

- In pure C-mode, incoming D bits are used to populate the cell's internal truth table, while overwritten truth table bits are output through the cell's D outputs; and
- In pure D-mode, the internal truth table of a cell is unchanged, and is used to map incoming D values to outgoing D and C values.

When C is allowed to take on values between 0 and 1, these behaviors are more complex:

- incoming D bits will be combined with the cell's pre-existing truth table bits, with the mixing ratio determined by the value of the C input;
- the combined value will replace the cell's truth table (according to some sort of pre-defined timing pattern, to be discussed below);
- the D outputs become a mix of (a) the D values specified by using the truth table as a lookup table (addressed by the incoming bits) and (b) the truth table entries themselves; and
- the C outputs become a mix of (a) the C values specified by using the truth table as a lookup table (addressed by the incoming bits) and (b) a 0;

Thus, if $C = 1$ or $C = 0$ the behavior is the same as with binary cells; but for $0 < C < 1$ the behavior is a mix of the two pure extremes.

The notion of a "truth table bit" requires clarification. Recall:

- A cell's truth table is in fact a continuous-valued *function with a continuous domain; and*
- there is no natural assignment of "bit numbers" to the function stored within a cell's memory.

In a binary cell, C-mode operation occurs bit-by-bit, with successive bits being read and written as time progresses. The goal however is simply to transfer the essence of a truth table via a narrow conduit (the D channel). In a Songline system, we serialize a cell's internal mapping function, and then transfer function values using time as a parameter for the serialized function. For a function of one variable ($y = f(x)$) with domain $[0, 1]$, we can do so by parameterizing the function using time as an index. For example, beginning at time $t = 0$ we can transmit the value of the function at 0, i.e., $f(0)$. Over the next second, we transmit the value of $f(t)$ as t increases from 0 to 1. After one second, the entire function will have been transmitted.

For a function of more than one variable, a single parameter (time) will not suffice for sweeping the entire domain. Instead we can define a scan pattern for sweeping the entire multi-dimensional domain with a single path, as shown in Figure 1.21. Unfortunately, this requires discretizing at least one dimension, at least for the simple solution presented here. It's is still being investigated whether something like a space-filling curve (Sagan 1994) might be used to map the higher-dimensional space to a one-dimensional parameterized space without such discretization.

1.5.2 C-Mode Applications and Benefits

Typically, the most common use of C-mode in a binary Cell Matrix is to read or write a cell's truth table, most often to copy one or more cells from one region to another. While this is also a use of C-mode in a Songline processor, there are other applications of C-mode as well. One is waveform generation: by loading a waveform into a cell's internal function memory and then placing the cell into C-mode, the pre-loaded waveform can be read out and fed to other parts of the system. This is an easy way, for example, to reproduce a fixed carrier wave for processing by an amplitude-modulation generator.

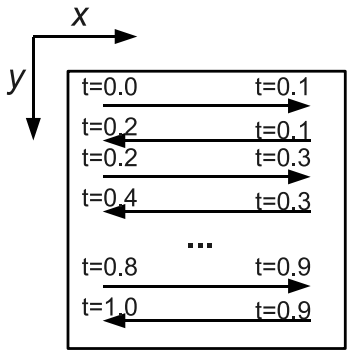

Fig. 1.21. Sample Scan Pattern for 2-D Domain. This figure shows the domain of a function $f(x, y)$ of two variables. The values of the function can be sampled from $t = 0$ to $t = 1$ in the order shown. Horizontal rows faithfully reproduce $f(x, y)$ for all values of x, but from row to row there is a discrete jump in the value of y.

Another interesting aspect of C-mode relates to function composition, and is illustrated in Figures 1.22 and 1.23. Each figure shows three cells: cells f and g are arbitrary "interesting" functions, while cell C is a cell whose job is to compose f with g. In Figure 1.22, cell C receives an input value x, sends it to cell f, receives $f(x)$, sends that to cell g, and receives $g(f(x))$. This provides an evaluation of $g(f())$ at the particular point x.

In Figure 1.23, cell C receives a "go" signal which places cell f into C-mode. Cell C reads cell f's function, sends it into cell g, reads back $g(f)$ (which is basically $g(f(t))$ at whatever time t has passed since cell f entered C-mode), and writes that new value back into cell f's function. After the entire function has been read and modified in this way, "go" is de-asserted, and cell f returns to D-mode. But now, the function stored in cell f is actually $g(f)$, i.e., the composite of functions f and g. $g(f(x))$ can subsequently be evaluated directly by sending x into cell f and reading back the value on the cell's D output.

Thus, on a Songline processor, there is a connection between C-mode and the notion of *operating on a function* (vs. operating on a function's value at a particular point). Thus we have a hardware implementation of a concept from *functional analysis* (Bachman 1966).

1.5.3 Advantages of a Songline Processor

While the notion of constructing circuits from elemental building blocks containing continuous-valued functions may seem archaic, there are a number of potential advan-

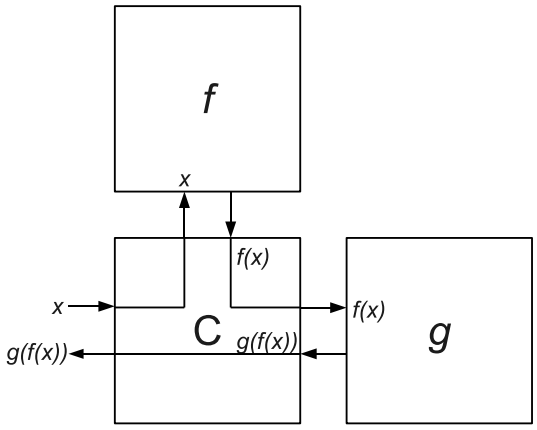

Fig. 1.22. Data-Based Function Composition. Cells f and g are used successively to evaluate first $f(x)$ and then $g(f(x))$.

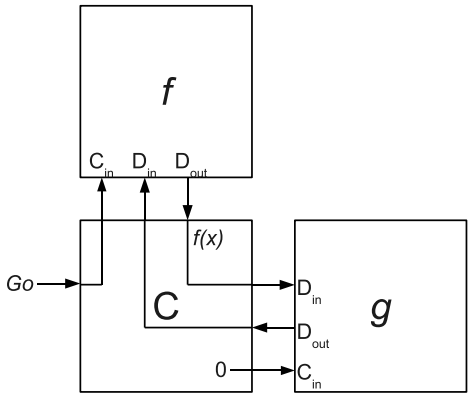

Fig. 1.23. C-Mode Based Function Composition. Cell f is reconfigured - with help from Cell g - to implement the composite function $g(f(x))$. This is done by using Cell g to evaluate g on *the function f itself*, i.e., to evaluate $g(f)$ vs. $g(f(x))$.

tages to this paradigm. One advantage is speed: an arbitrary function may be evaluated, in effect, by a single memory lookup. Evaluating a transcendental function takes no longer than simple multiplication by a constant. Moreover, such function lookups may be more precise then what can be achieved in a digital domain, where functions are approximated by (for example) part of their Taylor Series polynomial, which is then evaluated at a point near to (but generally not exactly equal to) the desired point of evaluation. This of course depends on the effectiveness of the implementation and specifically the storage mechanism for a cell's mapping function.

On the other hand, being analog in nature, there is perhaps an inherent fuzziness in the behavior of these cells, whose mapping might potentially be affected by temperature, pressure, or other environmental conditions. There is speculation that a certain degree of non-deterministic behavior may be beneficial in systems that attempt to mimic intelligent behavior (Pearn 2000). Thus a Songline processor may be an interesting platform for work in machine learning and artificial intelligence.

Finally, having a basic cell that can perform an arbitrary input-to-output mapping leads to a potentially simpler way to design circuits, as suggested by Figures 1.17-1.20. Such a system is also potentially simpler to interface with in a universe whose processes appear (at least on a macro scale) to be inherently analog and continuous-valued.

1.5.4 Significance

A Songline processor maintains the essential benefits of a binary Cell Matrix, including self-configurability, inherent defect tolerance, a lack of specialized components, versatile routine, and so on. Additionally, potential significance of the Songline architecture itself includes the following:

- it represents a reconfigurable approach to implementing analog circuits, similar in spirit to Field Programmable Analog Arrays (Kluwer 1998) but in some sense more direct;
- it provides a natural mechanism for mimicking analog behaviors, by storing such analog patterns directly into the function memory of a cell for later readback; and
- by using C inputs between 0 and 1, sampled analog data can be tweaked to a desired degree, providing opportunities for adaption based upon a model signal and a desired degree of variation.

This last point has potential applications to evolvable hardware systems (Greenwood and Tyrrell 2006). In its simplest form, a desired function can be developed by applying training data and comparing its actual output a against the desired output d. The difference $|a - d|$ can be used to generate a C input to the evolving cell. For a large difference, the evolving cell's C input will be large, and its truth table will tend to mimic the training data. As the difference decreases towards 0, the C input also diminishes, causing the cell's mapping function to change less and less. Mild or short-lived errors will causes relatively small changes in the mapping function, whereas more persistent errors will more-dramatically and more-continually shift the cell's function.

More generally, the Songline architecture is a very different approach to building circuits for processing information, and as such, the potential benefits (and pitfalls) won't be fully realized until it is explored further.

1.5.5 Implementation

Implementing a binary cell is extremely simple, requiring only a small digital memory, a counter, and a bit of logic to manage the cell's two modes. For a Songline processor whose cells store a continuous function, storage of that function is a much more difficult undertaking. Initial attempts have been based primarily on digitizing the domain and range of the function (as well as the inputs and outputs), and then using a standard digital memory. This is straightforward, but has two immediate drawbacks:

1. potential benefits unique to having a truly continuous-valued function are obviously lost, since the system is now basically again a digital circuit; and
2. the memory requirements for storing a function can be significant. For example, assuming 6-bit A-D/D-A converters and three-sided cells, each cell's function is stored in a truth table containing $2^{3\times 6}$ rows (since each side contributes 6 bits to the table lookup); each row contains 2×3 outputs (a D and C output on each of the 3 sides); and each output is itself 6 bits wide (since each analog value is stored as a 6-bit binary value). This means a single cell's truth table requires almost 10 million bits of storage. For an 8-bit A-D/D-A, this numbers grows to 8000 million bits; for 12 bit conversions it is close to 5 *trillion* bits. Thus even a modestly-accurate conversion requires a sizable memory.

It is therefore worth considering alternatives to simply digitizing the mapping function and the analog values processed in a Songline processor. One possible improvement is to continue to digitize the domain of the function, but to actually store analog values of the function at each (discrete) point in its domain. It may be feasible to implement such a system by exploiting the high density storage available on flash memory devices but to utilize the ability of each storage element's floating-gate to store an analog value (Welleknns and Van Houdt 2008).

It's worth noting that if we impose certain continuity requirements on stored functions, then the penalty for digitizing its domain may be reduced by incorporating circuitry for interpolating results. Also, note that the main difficulty in increasing the number of sides is only in the domain: for a cell with n sides, we simply implement $2n$ copies of our function storage mechanism within each cell (to drive a D and C output on each of the n sides).

The question of directly storing analog signals is not without precedent: early digital computers used analog memory modules such as mercury delay lines to store information in a series of acoustic waves that would travel the length of the tube, reach one end, be sampled, amplified, and re-transmitted to the other end (Eckert et al. 1971). A spring-based delay line (Figure 1.24) is another example based on the same principle. There are, however, complications that arise in trying to extract information from such a storage system. Suppose a time-varying signal is stored in a spring (for example).

Fig. 1.24. Mechanical Spring-Based Delay Unit. An analog signal can be sent into one end of this unit, where it is transformed into mechanical motion that causes the springs to oscillate. The signal travels through the spring to the other end, where a transducer converts the mechanical motion back into an electrical signal, which can be amplified and re-sent to the starting point. This is one way to store an analog signal.

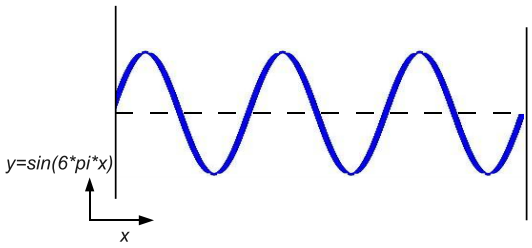

Fig. 1.25. Example of a trapped waveform. Here, the function $y = sin(6\pi x)$ has setup a displacement in the spring.

Figure 1.25 shows a trapped sine wave $y = sin(6\pi x)$ ($x\epsilon[0, 1]$). To evaluate this function f at any given point x, we only need to measure the displacement of the spring from its rest position at a point corresponding to x. Sometimes, this will be the point a distance x from the left of the spring (assuming the length of the spring is normalized to 1), as shown in Figure 1.26; but because the wave travels, the position of x changes, as shown, for example, in Figure 1.27.

So how do we measure this displacement? In a typical spring reverb system (such as is used for creating a reverberation effect with a guitar (Amplified Parts 2012), the end of the spring is connected to a transducer that transforms the spring's displacement into an electrical signal. It's thus possible to directly read the value of $f(x)$ for whatever value of x currently corresponds to the rightmost position on the spring. Ergo, assuming it takes 1 second for the wave to travel the entire length of the spring, we

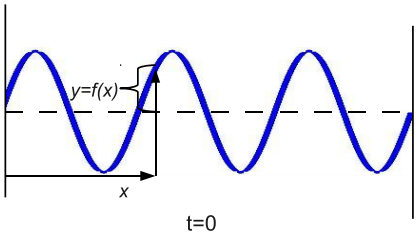

Fig. 1.26. Reading a Saved Value. To read the value of $f(x)$ at time $t = 0$, one needs to measure the displacement of the spring from its rest position at a distance x from the left side (where the signal is assumed to originate).

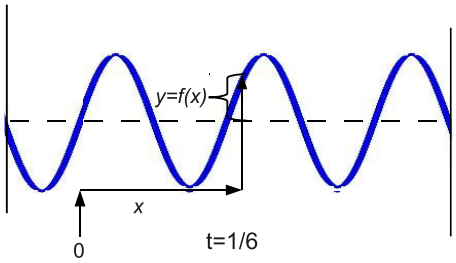

Fig. 1.27. Reading a Saved Value 1/6 Second Later. Since the trapped waveform travels down the spring, the position of "x" (and hence $f(x)$) will vary over time.

could read the value of $f(x)$ by simple waiting for time $t = 1 - x$ and then reading the displacement at the rightmost edge of the spring. This would give us an exact value, but with a time penalty of up to 1 second.

If a 1 second delay is unacceptable, we can instead decide to wait a maximum of $\frac{1}{2}$ second, and read the closest point available to the sensor anywhere during that $\frac{1}{2}$ second. Now we've reduced the time delay, but introduced a potential error in the value we read. More precisely, we will read the exact value of $f(x + \Delta x)$ where $|\Delta x| <= 0.5$.

An alternative is to add a second sensor in the middle of the spring (Figure 1.28). Now we can be assured that the point x will pass *some* sensor in no more than $\frac{1}{2}$ second. In this setup though we have another option: we can immediately read the value at whichever sensor is closest to the point x, and thus obtain *without delay* an *approximation* to $f(x)$. Or we can accept a delay somewhere between 0 and $\frac{1}{2}$ an a possibly smaller error in where we evaluate the function. More-generally, for n evenly-

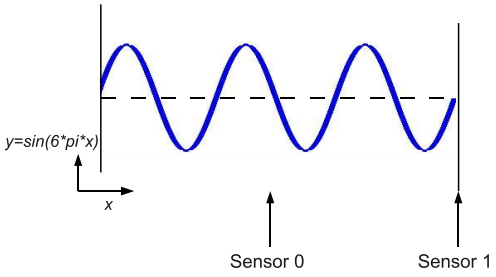

Fig. 1.28. Spring-Based Storage System with a Second Sensor. Adding this second sensor (in the middle of the unit) allows $f(x)$ to be sampled with a delay of less than 0.5 sec.

spaced sensors, we can read with a delay of 0 but an error of $\frac{1}{n}$, or we can wait up to $\frac{1}{n}$ seconds and read an exact value. The more sensors, the better our situation. But for any finite number of sensors, we do not have the option of eliminating both the delay in reading a given function value and the potential error in that reading. Sampling at a precisely desired point in time and space is not possible: we can decrease the error in one, but only at the expense of increasing the error in the other. This is perhaps reminiscent of other natural phenomena (Heisenberg 1927), though it's unclear what the actual connection might be.

All of the above is only for a function of one variable $f(x)$. The question of two variables is more complex. Visually, an image of a deformed rubber sheet may be useful: at any point (x, y) the displacement of the sheet corresponds to the function's value $f(x, y)$. If this deformation is setup as a traveling wave, then a situation similar to the one-dimensional case arises, and we could populate the domain with an array of sensors capable of reading the sheet's displacement at certain predefined points. Alternatively, if the sheet is rigidly deformed, a series of movable sensors might be used to take a height reading at a desired point. Here again, we can read without delay by approximating the point (x, y); or we can introduce a delay as a sensor is repositioned and then used to read the value exactly.

A different model for storing a function $f(x, y, z)$ might be to use the temperature at a point (x, y, z) within a three-dimensional region of space as a coding for the function value $f(x, y, z)$. Of course, creating a system to do this (and to maintain the temperature values) is likely impossible, but it suggests the general idea of mapping the domain to 3-space and reading some physical property of each point (x, y, z) in space to discern the value of $f(x, y, z)$ at that point. Another idea that comes to mind is, perhaps, to code information in the phase of light at each point in a 3-D region: perhaps something along the lines of creating and reading a hologram. It's not currently clear how a function $f(x, y, z, w)$ of four variables might be stored physically. To date, most work on Songline processors has been done with three-sided cells.

Yet another alternative for storing functions of more than one variable is (again) to consider a space-filling curve (or an approximation to it) to parameterize a 2-, 3- or 4-dimensional region, mapping it to a one-dimensional region (in a way similar to Figure 1.21) and using that mapping to locate desired points. Of course, this requires certain continuity assumptions on the function being stored, and we are in effect digitizing the domain. But as an approximation, it does provide a way to store functions of higher-dimensional domains using a one-dimensional storage system.

Finally, it should be noted that the C-mode component of a cell's operation - in which the function itself is interrogated and read/written - such parameterization is necessary (as described previously), in which case using this approach to store the function may have its own benefits in terms of consistency of the cell's function.

1.5.6 Songline Processor Prototype

Figure 1.29 shows a prototype of a Songline processor. The dials are used to generate analog inputs, while the meter reads the analog output of one of several channels. Software on a host machine acts as the actual processor (an 8x8 array of 3-sided cells). A compiler converts desired cell behavior (expressed in a C-like syntax) into the internal structures necessary, based on 6-bit digitizing of analog values. A configuration file designates certain edge cells' inputs and outputs as corresponding to particular channels, which the I/O box (shown in the figure) can write and read. Additional connection points are available on the back of the unit, for connection of channels to function generators, oscilloscopes, and so on.

Using this system, initial designs have been developed and tested, including basic amplifiers, differentiators, flip flops, and so on. Cell replication from a source to a target using C-mode has also been successfully demonstarated. The system works well, though the differences from a typical digital system are often surprising (such as the wide range of effects race conditions can exhibit).

Next steps include:

- continuing to develop circuits on this test bed;
- experimenting with single-variable function storage using a spring-based delay system;
- looking into the feasibility of exploiting floating-gate technology as a means of storing and retrieving real-valued quantities;
- trying to better understand the time vs. error tradeoffs of incorporating multiple sensors in the function readback system; and
- looking further into holographic techniques to see if there's some way to use holography to store functions of 2 or 3 variables.

1.6 Status and Future Work

1.6.1 Current Status

Several specifications for the Cell Matrix architecture have been designed (Macias et al. 1999; Durbeck and Macias 2001c; Macias and Raju 2001), and these different

Fig. 1.29. Songline Processor Prototype. Analog inputs are specified with the knobs, while analog outputs can be read from the meter (the selector below the meter chooses which output channel is displayed). Software running on a USB-connected host machine emulates an 8×8 array of 3-sided cells. 6-bit digitizing is used, resulting in a total function memory of 128 MB.

implementations have been used in a number of software simulators (Cell Matrix Corporation 2006b). Most work to date has been done using software simulators.

Hardware implementations have also been developed, including a small custom ASIC implementation. More recent implementations have used traditional FPGAs to implement the Cell Matrix architecture (Macias and Durbeck 2004; Durbeck and Macias 2001a, 2002; Macias and Durbeck 2005b), including a self-contained 8×8 Cell Matrix board called the Mod-88 (Macias and Durbeck 2005b), a 4×4 array of which has been placed on the web for use (Cell Matrix Corporation 2006c).

A complete set of tools has been constructed and placed on the web to permit anyone to construct Cell Matrix circuits and debug them using a graphical layout editor and libraries of already-built components (Cell Matrix Corporation 2006b). A place and route tool was recently developed to automatically generate layouts from circuit descriptions (Macias 2006).

1.6.2 Some Applications of Self-Configurability

There are a number of areas where this shift from external- to internal-control appears especially beneficial to the design of computational and processing systems. There has been and continues to be impetus to scale systems to greater numbers of components and greater levels of complexity in the tasks they undertake. As the system's size and complexity are scaled up, the difficulties associated with managing and maintaining the

system also increase. When rapid increases in scaling eventually occur (Vinge 1993), it may no longer be practical to continue scaling up current strategies. Instead, new approaches to managing extreme complexity may be required.

Utilizing internal control mechanisms is a better strategy than today's external system control as computers become more complex. A key difficulty in managing extremely complex systems is the dependence on a single, centralized unit for all management tasks. In contrast, one can distribute the management and control of the system among a large number of separate management units, thereby reducing the load on each management unit, while also improving the proximity between each management unit and the circuits that it is managing. Care must be taken to avoid introducing new complexities as the number of management units is itself scaled up.

A possible solution is to use the massive resources of such large-scale systems to solve the very problem being caused by their size, i.e., tackle the problems of large-scale system design by using a large-scale system. Some example areas where this may be applied are:

- Manufacturing Defects: In order to utilize many orders of magnitude more devices, computational systems such as CPUs and FPGAs will need to be able to use imperfect hardware, because it will be too difficult and expensive to build perfect hardware. It thus seems there will need to be a design shift, from systems that are completely free of defects to systems that can handle such defects. One way to approach this challenge is to have the system itself perform initial checks on its own hardware, testing its subsystems in an efficient (parallel) manner, and noting defective regions in a way that subsequent processing can use to avoid those defects.
- Run-time Defects: Even in systems that are manufactured perfectly, or whose defects have been worked-around, there are still run-time defects, i.e., temporary or permanent errors that occur while the system is operating. Given the large number of components expected in future systems, the job of monitoring them for proper functioning cannot reasonably be handled from a single, centralized (i,e., external) location. Instead, the job of defect detection and mitigation may better be handled from within the system, using the large number of system components as a resource for handling the complexity of this task.
- System Design: With a jump to Avogadro-scale systems, one could expect system design times to become prohibitively long. One example of applying our thesis to this problem would be to use a massive FPGA to implement and run massively-parallel CAD tools, which are then used for subsequent design work. Another is to decouple system design from system construction, so that system construction happily continues to march down Moore's Law curve, while the increasingly complex task of systems design has time and opportunity to develop as resources are made available.
- Initial System Configuration/Bootstrap: As systems scale to use many orders of magnitude more devices, the problem of configuring or bootstrapping them (the process of specifying their initial setup in the case of such soft hardware as FPGAs, and the bootstrapping process of invoking and initializing all processes that constitute the running system in the case of all computation systems including FPGAs and computers) rapidly becomes worse, until configuration and initialization

times may be so long that systems cannot even complete initialization. Again, a possible solution is to use the massively-parallel system itself as the control system used to implement a very powerful, parallel bootstrap system.

1.6.3 Possible Manufacturing Options

Because even simple circuitry implemented on the Cell Matrix tends to consume a large number of cells, practical hardware Cell Matrices are difficult to create using conventional manufacturing techniques. There are, however, a number of conceivable approaches to manufacturing large Cell Matrices ("large" means a Cell Matrix containing a large number of cells).

Aggressive Silicon Techniques

One approach is to use aggressive techniques in silicon, such as deep deep sub-micron technology, while taking advantage of the fault-handling capabilities of the Cell Matrix to manage the inevitably-high defect count. This, of course, requires a mitigation technique that costs less (in terms of area/cell count) than what is gained by using a very aggressive fabrication technology.

An added benefit to using cutting-edge technologies to manufacture a Cell Matrix is the possibility of using the Cell Matrix itself as a *process driver*, i.e., as a means of debugging the fabrication process itself (Durbeck and Macias 2002). Because a Cell Matrix has an inherent introspection capability, it can be used to analyze the characteristics of the manufactured cells within itself, including things such as their logical behavior, the speed of their operation, the pattern of defective cells' locations, and so on. Because pathways from cell to cell are built out of cells, there are generally multiple pathways from any cell to any other cell. This means that even regions containing a large number of defects might be thoroughly analyzed (Durbeck and Macias 2002).

Wafer-Scale Integration

Another potential approach to manufacturing large Cell Matrices is to employ wafer-scale integration (WSI) (Saucier and Trilhe 1986; Wyatt and Raffel 1989; Fuchs and Swartzlander Jr 1992; IEEE 1995; Zeng et al. 2005), where, instead of dicing a wafer into a number of individual chips, the entire wafer is used to implement a single circuit. WSI has been explored as a means to overcome die-level defects, and in general has to address the problem of a wafer typically containing some defective die (e.g., (Boubekeur et al. 1992; Saucier et al. 1988)). Various special-purpose architectures have been developed for WSI to allow operation on top of imperfect wafers (e.g., (Boubekeur et al. 1992; Saucier et al. 1988)), as well as a general-purpose methodology for using wafer-scale fabrication (Alam et al. 2002).

Of course, since a Cell Matrix is inherently a fault-isolating architecture, and since faults can be detected and managed efficiently using some of the techniques described above, it is an ideal architecture for implementing using WSI.

Three-Dimensional Fabrication

While most discussion of the Cell Matrix architecture in this chapter has focused on two-dimensional matrices, it is perfectly feasible to create the three-dimensional Cell

Matrix. A three-dimensional Cell Matrix is actually much more powerful than a two-dimensional one, for a number of reasons:

- each cell is more powerful, being a 6-input 6-output device (assuming a cube-based structure/topology);
- circuit routing is easier, at least for two-dimensional circuits, since, being embedded in a higher-dimensional space, non-adjacent components can be connected via the third dimension;
- for a given number of components, the maximum path length, and hence maximum delay, can be greatly decreased — for example, a square circuit containing a trillion cells would be $1,000,000 \times 1,000,000$ cells, giving a maximum corner-to-corner path length of $2,000,000$ cells; while a cubic circuit containing a trillion cells would be $10,000 \times 10,000 \times 10,000$, giving a maximum corner-to-corner path length of only $30,000$ cells;
- configuration of a three-dimensional circuit can be much faster than configuration of a two-dimensional circuit with the same cell count; in the above example, the two-dimensional case would require $2,000,000$ operations ($1,000,000$ to construct one row of Supercells, and another $1,000,000$ for each Supercell to create a column of Supercells), while the three-dimensional case would require only $30,000$ operations ($10,000$ to make a row of Supercells; another $10,000$ to create a column of $10,000$ Supercells below each of those, thus giving a two-dimension plane of Supercells; and a final $10,000$ operations for each of *those* $100,000,000$ Supercells to create a line of Supercells in the Z-axis);
- a three-dimensional matrix can be treated as a series of two-dimensional matrices with inter-matrix access in the Z dimension, thus enabling techniques such as rapid context switching, parallel reading of cells, parallel writing of cells, plane-to-plane voting, and so on; and
- a three-dimensional matrix has a number of two-dimensional surfaces available to the outside world, thus affording a high-bandwidth mechanism for parallel input and output of data to and from the matrix.

There have been various attempts by researchers to create three-dimensional circuitry (Seeman 1982; J. DePreitere 1994; Alexander et al. 1995; Borriello et al. 1995; Meleis et al. 1997; Leeser et al. 1997). One problem often encountered is heat buildup, but this need not be an issue with a massively parallel architecture such as the Cell Matrix, since the idea is to get algorithm speedup through the use of massively-parallel algorithms, rather than through raw uni-processor speed. Another significant impediment for three-dimensional fabrication is, again, the manufacturing defect rate, but this can be (to some degree) mitigated using Cell Matrix fault handling techniques.

1.6.4 Nanotechnology

Any discussion of high-density, three-dimensional fabrication inevitably leads to the topic of Nanotechnology (Montemerlo et al. 1996; Kamins and Williams 2001; Stan et al. 2003). Roughly speaking, nanotechnology is the science of atomic-scale manufacturing. In the context of using nanotechnology to manufacture Cell Matrices, our

primary interest is not so much in the size of the manufactured cells per se,but rather in the extremely high cell count that can be achieved because of that size. That is, our interest is not in making small Cell Matrices, but rather in creating Cell Matrices containing a huge number of cells. This might involve manufacturing cells out of logic gates that themselves are comprised of a small number (say 10-$1,000$) of atoms, single electrons, etc., with each resulting cell containing fewer than a million atoms. At such a scale, we can talk about quantities such as one *mole* of cells, i.e., a number of cells equal to Avogadro's numbers, or roughly 10^{23} cells. Note that a three-dimensional Cell Matrix containing 10^{23} cells would have only $100,000,000$ cells along each edge.

1.6.5 Cell Matrix Support for Nanotechnology

While the Cell Matrix architecture stands to benefit a great deal from a true atomic-scale fabrication technologies, it is also possible that such technologies may also benefit from the Cell Matrix. For example, in trying to manufacture three-dimensional circuitry,one often manufactures a series of two-dimensional layers using conventional techniques (e.g., lithographic techniques), and then stacks these layers in the third dimension. While the former process is usually quite precise, the latter is not: it involves the manipulation of macro-scale objects, such as individual silicon die or silicon wafers (Fraunhofer Institute for Reliability and Microintegration, Munich 2006; Misc 2006; IEEE 1995; Ababei et al. 2004).

However, if the layers being stacked are actually Cell Matrices, the cells themselves can be used to create circuits on each layer that investigate their own placement relative to the layers above and below them. By staggering the placement of the cells within each layer, that is, using non-uniform spacing between cells, it may be possible to guarantee that some of the cells will align between layers (while also guaranteeing that some cells will not). By voluntarily sacrificing some of the cells, we can insure that some of the cells will create inter-layer connections. Circuits can then be constructed to discover where these connections have been made, and that information used in the configuration of subsequent circuits.

1.6.6 Other Approaches to Manufacturing

When thinking about manufacturing a Cell Matrix, it is worthwhile to consider approaches that may lack characteristics typically required for conventional circuit manufacture. Some of the characteristics that differ for a Cell Matrix manufacturing process vs. conventional circuits include:

- speed — a Cell Matrix is not required to have extremely fast cells, since one may hope to obtain algorithm speedup via massive parallelism rather than raw component speed;
- power dissipation — because, again, the individual cells can be run slowly, with overall speedup achieved via parallelism;
- reliability — as we have seen, perfect manufacture is not strictly necessary for the creation of a viable Cell Matrix: a certain degree of defects is acceptable; and

- physical size — the primary requirement for a useful Cell Matrix is not that it be physically small, but rather that it contain a huge number of cells.

Taking these considerations into account, there are some unconventional possibilities for manufacturing a Cell Matrix. One possibility is to use printable circuit technology (Plastic Logic 2006; Burns et al. 2004; Sirringhaus et al. 2006; MacDonald 2006; Wong et al. 2006) to create two-dimensional sheets of cells. These sheets could be folded and stacked, to create narrow, but arbitrarily-long matrices. Alternatively, single sheets could be stacked, and connectors pierced through the sheets to create inter-sheet connections. Even if only some cells are connected from sheet-to-sheet, this would still offer some of the benefits of a fully three-dimensional matrix.

This also leads to the notion of trying to weave a matrix, utilizing some of the ideas being used for the design of smart clothing (Cakmakci and Koyuncu 2000; Cakmakci et al. 2001; Martin 2006; Edmison et al. 2002; Martin et al. 2003; Marculescu et al. 2003). If cells can be constructed simply by controlling the pattern of a weaving, then, again, arbitrarily-long two-dimensional sheets could be manufactured. This approach is worth considering if only because the manufacture of textiles is one of the oldest manufacturing processes in human history, and is estimated to date back 12,000 years (Sabalan Group 2006).

Because of the regular structure of the Cell Matrix, it may be possible to exploit natural processes in its manufacture: for example, the process of crystal growth. Of course, this would require a means for associating logic circuits with certain types of crystals, arranging things so that as the crystal grows, so does the matrix. A more-controlled approach is being taken in the use of DNA as a scaffolding for the placement of carbon nanotubes (Seeman 1982; Winfree 1998; Winfree et al. 1998; Dwyer et al. 2004a,b; Patwardhan et al. 2004; Park et al. 2006; Pistol et al. 2006; Patwardhan et al. 2006; Kim et al. 2004; Winfree and Bekbolatov 2004; Seeman 2003; Rothemund et al. 2004; Robinson and Seeman 1987; Winfree 2003), which could be applied to the construction of Cell Matrix cells.

1.6.7 CAD Issues — Magic Polygons

It remains unanswered how best to represent self-modifying circuitry such as that which can be implemented on the Cell Matrix. One idea is to use the notion of *serialization*, wherein a collection of cells is used to generate a stream of binary data corresponding to it. Once a circuit has been serialized, it can be processed using standard digital circuits: it can be stored, retrieved, steered through circuitry via mux/demux logic, compared with other streams, and so on. Along with serialization is the process of *de-serialization*, wherein a stream corresponding to a circuit is used to configure a set of cells to implement that circuit.

Figure 1.30.a shows an example of this serialization/de-serialization process. A sample source circuit is shown on the right, surrounded by a dashed box called a **"Magic Polygon,"** which indicates a region of the circuit that is to be serialized. The line coming from the box transfers this stream of ordinary binary data to the de-serializer on the left, where a new copy of the circuit is created in Figure 1.30.b.

The "*" inside each polygon is used to position de-serialized circuitry relative to the original serialized circuit. The GO signal indicates initiation of the de-serialization operation. Figure 17 shows a simple circuit replication.

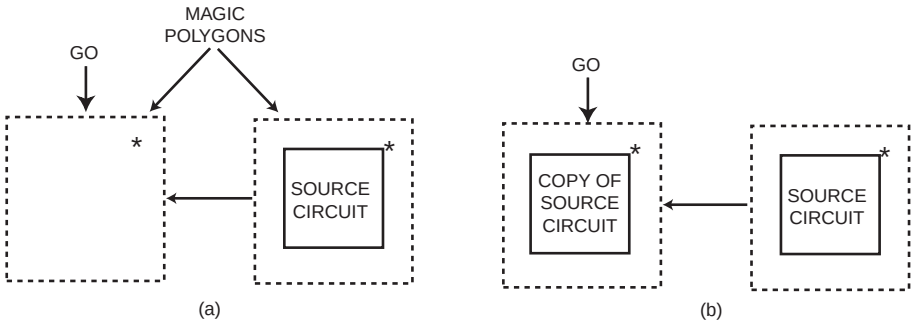

Fig. 1.30. Serializaton of a Source Circuit. In (a), the truth tables comprising the Source Circuit are transmitted along the wire to the de-serializer on the left. "*" is an anchor point for positioning the new circuit. In (b), the de-serializer has synthesized a copy of the Source Circuit using the serial bit stream it receives.

Figure 1.6.7 shows an example of a simple Hardware Library, where one of four circuits can be selected for synthesis. The bitstreams for each circuit are sent into the 4-1 selector, which chooses one of them for synthesis by the Magic Polygon to the South.

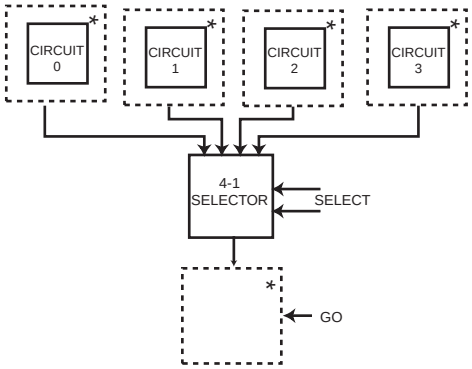

Fig. 1.31. Example of a Hardware Library. The 4-1 Selector can choose the bitstream corresponding to one of four circuits. The selected circuit will be synthesized to the South.

Figure 1.32 shows a more-complex example. In this case, the Magic Polygon on the right itself contains a Magic Polygon. This means the synthesized circuit will also contain a Magic Polygon. When the circuit in Figure 1.32.a is serialized and then de-serialized, the circuit of Figure 1.32.b results. This looks exactly like Figure 1.32.a,

except that the East-to-West wire has been extended. Each time the circuit's bitstream is de-serialized, the wire is extended another step. This can be used as a mechanism for synthesizing wires, as described above in Section 1.3. Note that in this example, the GO line has not been shown. The Magic Polygon on the left needs a GO signal in order to initiate its de-serialization process, but since the location of this Magic Polygon changes, the line driving the GO input also needs to be extended at each step. For simplicity, this and other details have been excluded from Figure 1.32.

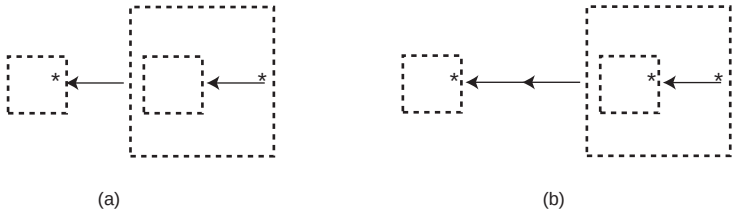

Fig. 1.32. Magic Polygon Representation of an Extendible Wire. (a) shows the initial setup. (b) shows the result after deserialization and reserialization: the wire has been extended.

While Magic Polygons are an easy way to visualize the serialization/de-serialization process, they are more applicable to circuit schematics than to, say, a Hardware Definition Language (HDL) description of a circuit. Ongoing research efforts seek to elaborate on the details of Magic Polygons, to extend them to the realm of HDLs, and to develop tools for implementing their functional behavior.

References

Ababei, C., Maidee, P., and Bazargan, K. (2004). Exploring potential benefits of 3D FPGA integration. *Field-Programmable Logic and its Applications*, pages 874–880.

Abdi, H. (1994). A neural network primer. *Journal of Biological Systems*, 2(3):247–283.

Alam, S., Troxel, D., and Thompson, C. (2002). A Comprehensive Layout Methodology and Layout-Specific Circuit Analyses for Three-Dimensional Integrated Circuits. *ISQED International Symposium on Quality Electronic Design, 2002*, page 246.

Alexander, M., Cohoon, J., Colflesh, J., Karro, J., and Robins, G. (1995). Three-dimensional field-programmable gate arrays. *ASIC Conference and Exhibit, 1995., Proceedings of the Eighth Annual IEEE International*, pages 253–256.

Amplified Parts (2012) Spring Reverb Tanks Explained and Compared. *http://www.amplifiedparts.com/tech_corner/spring_reverb_tanks_explained_and_compared*, retrieved November 2012.

Arms, K. and Camp, P. (1987). *Biology*. Saunders Philadelphia, USA, third edition edition.

Aspray, W. and Burks, A. (1987). Papers of John von Neumann on Computing and Computer Theory, volume 12 of Charles Babbage Institute Reprint Series for the History of Computing.

Bachman, G. and Narici, L. (1966) Functional Analysis. Academic Press.

Borriello, G., Ebeling, C., Hauck, S., and Burns, S. (1995). The Triptych FPGA architecture. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 3(4):491–501.

Boubekeur, A., Patry, J., Saucier, G., and Trilhe, J. (1992). Configuring a Wafer-Scale Two-Dimensional Array of Single-Bit Processors. *Computer*, 25(4):29–39.

Burns, S., Kuhn, C., Jacobs, K., MacKenzie, J., Ramsdale, C., Arias, A., Watts, J., Etchells, M., Chalmers, K., Devine, P., et al. (2004). Printing of polymer thin-film transistors for active-matrix- display applications. *Journal of the Society for Information Display*, 11:599.

Cakmakci, O. and Koyuncu, M. (2000). Integrated electronic systems in flexible and washable fibers. *None*. filed with the United States Patent Office and the European Patent Office.

Cakmakci, O., Koyuncu, M., and Eber-Koyuncu, M. (2001). Fiber Computing. *Proc. of the Workshop on Distributed and Disappearing User Interfaces in Ubiquitous Computing, CHI*.

Cell Matrix Corporation (2006a). bibliography for Cell Matrix-related research. http://www.cellmatrix.com/entryway/products/pub/bibliography.html.

Cell Matrix Corporation (2006b). Cell Matrix Software. http://www.cellmatrix.com/entryway/products/software/software.html.

Cell Matrix Corporation (2006c). MOD 88 Online Viewer. http://cellmatrix.dyndns.org:12001/cgi-bin/mod88/obs2.cgi?

Chatwin, B. (1986). *The Songlines*. Penguin Books.

Darwin, C. (1859). *The Origin of Species by Means of Natural Selection. Or the Preservation of Favoured Races in the Struggle for Life*. Murray, London, United Kingdom.

Deutsch, L. and Schiffman, A. (1984). Efficient implementation of the smalltalk-80 system. *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302.

Duncan, R. (1989). Design goals and implementation of the new High Performance File System. *MICROSOFT SYST. J.*, 4(5):1–14.

Durbeck, L. and Macias, N. (2001a). Autonomously Self-Repairing Circuits. NASA SBIR Phase II Proposal.

Durbeck, L. and Macias, N. (2001b). Autonomously Self-Repairing Circuits. NASA SBIR Phase I Final Report.

Durbeck, L. and Macias, N. (2001c). Self-configurable parallel processing system made from self-dual code/data processing cells utilizing a non-shifting memory. US Patent 6,222,381.

Durbeck, L. and Macias, N. (2001d). The Cell Matrix- An architecture for nanocomputing. *Nanotechnology*, 12(3):217–230.

Durbeck, L. and Macias, N. (2002). Defect-tolerant, fine-grained parallel testing of a Cell Matrix. *Proceedings of SPIE ITCom*, 4867.

Dwyer, C., Johri, V., Patwardhan, J., Lebeck, A., and Sorin, D. (2004a). Design tools for self-assembling nanoscale technology. *Nanotechnology*, 15:1240–5.

Dwyer, C., Poulton, J., Taylor, R., and Vicci, L. (2004b). DNA self-assembled parallel computer architectures. *Nanotechnology*, 15(11):1688–1694.

Eckert, J. P. et al. (1971) The UNIVAC System. reprinted in *Computer Structures: Readings and Examples*. McGraw-Hill.

Edmison, J., Jones, M., Nakad, Z., and Martin, T. (2002). Using piezoelectric materials for wearable electronic textiles. *Wearable Computers, 2002.(ISWC 2002). Proceedings. Sixth International Symposium on*, pages 41–48.

Fischer, T. (1987). Heavy-ion-induced, gate-rupture in power MOSFETs. *Nuclear Science, IEEE Transactions on*, 34(6):1786–1791.

Fraunhofer Institute for Reliability and Microintegration, Munich (2006). Department of Si Technology and Vertical System Integration. http://www.izm-m.fraunhofer.de/files/fraunhofer2/si-technology_vsi.pdf, accessed 10/31/2006.

Fuchs, W. and Swartzlander Jr, E. (1992). Wafer-Scale Integration: Architectures and Algorithms. *Computer*, 25(4):6–8.

Greenwood, G. and Tyrrell, A. (2006). Introduction to Evolvable Hardware. Wiley-IEEE Press.

Haldane, J. (1931). The Philosophical Basis of Life.

Heisenberg, W. (February 1927). Werner Heisenberg, in a letter to Wolfgang Pauli.

IEEE (1989 - 1995). Proceedings of the international conference on wafer scale integration.

J. DePreitere, e. a. (1994). An Optoelectronic 3D Field Programmable Gate Array. In Hartenstein, W. and Servit, M., editors,*Field-Programmable Logic: Architectures, Synthesis, and Applications, Lecture Notes in Computer Science*, volume 849. Springer-Verlag Berlin, Germany.

Kamins, T. and Williams, R. (2001). Trends in nanotechnology: Self-assembly and defect tolerance. *Proc. NSF Partnership in Nanotechnology Conf.*

Kauffman, S. (1993). *The Origins of Order: Self-organization and Selection in Evolution*. Oxford University Press.

Kim, J., Hopfield, J., and Winfree, E. (2004). Neural network computation by in vitro transcriptional circuits. *Advances in Neural Information Processing Systems*, 17:681–688.

Kluwer (1998). Analog Integrated Circuits and Signal Processing. Special Issue on Field-Programmable Analog Arrays, Vol 17 Numbers 1-2.

Koza, J. (1992). *Genetic Programming: on the programming of computers by means of natural selection*. Bradford Book.

Leeser, M., Meleis, W., Vai, M., and Zavracky, P. (1997). Rothko: A three dimensional FPGA architecture, its fabrication, and design tools. *Seventh International Workshop on Field Programmable Logic and Applications*.

Lennox, J. (2001). *Aristotle's Philosophy of Biology: Studies in the Origins of Life Science*. Cambridge University Press.

MacDonald, W. A. (2006). Advanced Flexible Polymeric Substrates. In Klauk, H., editor, *Organic electronics: Materials, manufacturing & its applications*. Wiley.

Macias, N. (1999). The PIG Paradigm: The Design and Use of a Massively Parallel Fine Grained Self-Reconfigurable Infinitely Scalable Architecture. *Proceedings of The First NASA/DOD Workshop on Evolvable Hardware (EH'99)*.

Macias, N. (2001). 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.

Macias, N. (2006). Cell Matrix Place and Route Tool: Changes and Improvements. White Paper delivered to Los Alamos National Laboratory under sub-contract #90843-001-04 4x.

Macias, N. and Durbeck, L. (2002). Self-assembling circuits with autonomous fault handling. *Evolvable Hardware, 2002. Proceedings. NASA/DoD Conference on*, pages 46–55.

Macias, N. and Durbeck, L. (2004). Adaptive methods for growing electronic circuits on an imperfect synthetic matrix. *Biosystems*, 73(3):172–204.

Macias, N. and Durbeck, L. (2005a). unpublished white papers and talks delivered to Los Alamos National Laboratory under sub-contract #90843-001-04 4x.

Macias, N. and Durbeck, L. (2005b). A Hardware Implementation of the Cell Matrix Self-Configurable Architecture: The Cell Matrix MOD 88. *Evolvable Hardware, 2005. Proceedings. 2005 NASA/DoD Conference on*, pages 103–106.

Macias, N., Henry III, L., and Raju, M. (1999). Self-reconfigurable parallel processor made from regularly-connected self-dual code/data processing cells. US Patent 5,886,537.

Macias, N. and Raju, M. D. (2001). Method and Apparatus for Automatic High-Speed Bypass Routing in a Cell Matrix Self-Configurable Hardware System. US Patent 6,577,159.

Mange, D., Sipper, M., Stauffer, A., and Tempesti, G. (2000). Toward self-repairing and self-replicating hardware: the Embryonicsapproach. *Evolvable Hardware, 2000. Proceedings. The Second NASA/DoD Workshop on*, pages 205–214.

Marculescu, D., Marculescu, R., Zamora, N., Stanley-Marbell, P., Khosla, P., Park, S., Jayaraman, S., Jung, S., Lauterbach, C., and Weber, W. (2003). Electronic textiles: a platform for pervasive computing. *Proceedings of the IEEE*, 91(12):1995–2018.

Martin, T. (2006). Tom Martin's Wearable Electronic Textiles research group at Virginia Tech. http://www.ccm.ece.vt.edu/etextiles/, http://www.ccm.ece.vt.edu/etextiles/publications/ accessed 10/31/2006.

Martin, T., Jones, M., Edmison, J., and Shenoy, R. (2003). Towards a design framework for wpearable electronic textiles. *Wearable Computers, 2003. Proceedings. Seventh IEEE International Symposium on*, pages 190–199.

Meleis, W., Leeser, M., Zavracky, P., and Vai, M. (1997). Architectural Design of a Three Dimensional FPGA. *Advanced Research in VLSI, 1997. Proceedings., Seventeenth Conference*, pages 256–268.

Misc (2006). International Journal of Chip-Scale Electronics, Flip-Chip Technology, Optoelectronic Interconnection and Wafer-Level Packaging. http://www.chipscalereview.com accessed 10/31/2006.

Montemerlo, M., Love, J., Opiteck, G., Goldhaber-Gordon, D., and Ellenbogen, J. (1996). Technologies and Designs for Electronic Nanocomputers. *The MITRE Corporation, McLean, VA, MITRE Tech. Rep. MTR 96W0000044, July*.

Ortega-Sanchez, C., Mange, D., Smith, S., and Tyrrell, A. (2000). Embryonics: A Bio-Inspired Cellular Architecture with Fault- Tolerant Properties. *Genetic Programming and Evolvable Machines*, 1(3):187–215.

Page, I. (1996). Constructing hardware-software systems from a single description. *The Journal of VLSI Signal Processing*, 12(1):87–107.

Park, S., Pistol, C., Ahn, S., Reif, J., Lebeck, A., Dwyer, C., and LaBean, T. (2006). Finite-size, Fully-Addressable DNA Tile Lattices Formed by Hierarchical Assembly Procedures. *Angewandte Chemie*, 45:735–739.

Patwardhan, J., Dwyer, C., Lebeck, A., and Sorin, D. (2004). Circuit and System Architecture for DNA-Guided Self-Assembly of Nanoelectronics. *Foundations of Nanoscience: Self-Assembled Architectures and Devices. Proceedings 2004*, pages 344–358.

Patwardhan, J., Dwyer, C., Lebeck, A., and Sorin, D. (2006). NANA: A nano-scale active network architecture. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 2(1):1–30.

Pearn, J. (2000). Email conversation with N. Macias, May 2000. http://www.artificialbrains.com

Pistol, C., Lebeck, A., and Dwyer, C. (2006). Design automation for DNA self-assembled nanostructures. *Proceedings of the 43rd annual conference on Design automation*, pages 919–924.

Plastic Logic (2006). Plastic Logic, developer of printed flexible thin film transistor (TFT) arrays. http://www.plasticlogic.com/technology.php accessed 10/31/2006.

Prodan, L., Tempesti, G., Mange, D., and Stauffer, A. (2003). Embryonics: Electronic Stem Cells. In Abbass, H., Standish, R., and Bedau, M., editors, *Artificial Life VIII: Proceedings of the Eighth International Conference on Artificial Life*, pages 101–105. Bradford Book.

Robinson, B. and Seeman, N. (1987). The design of a biochip: a self-assembling molecular-scale memory device. *Protein Engineering Design and Selection*, 1:295–300.

Rothemund, P., Papadakis, N., and Winfree, E. (2004). Algorithmic self-assembly of DNA Sierpinski triangles. *PLoS Biology*, 2(12):2041–2053.

Sabalan Group (2006). Textile History. http://www.sabalangroup.com/aboutus-history-textilehist-en.html.

Sagan, H. (1994) Space-Filling Curves. Springer-Verlag.

Saha, C., Bellis, S., Mathewson, A., and Popovici, E. (2004). Performance Enhancement Defect Tolerance in the Cell Matrix Architecture. In *Proceedings of MIEL 2:2004*, pages 777–780.

Saucier, G., Patry, J., and Kouka, E. (1988). Defect tolerance in a wafer scale array for image processing. *Proc. Int'l Workshop Defect and Fault Tolerance in VLSI Systems, Univ. of Massachusetts, Amherst, Oct.*, 8:8.2–1–8.2–13.

Saucier, G. and Trilhe, J. (1986). *Wafer scale integration*. North-Holland.

Schmit, H. (1997). Incremental reconfiguration for pipelined applications. *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 47–55.

Seeman, N. (1982). Nucleic acid junctions and lattices. *J Theor Biol*, 99(2):237–47.

Seeman, N. (2003). Biochemistry and structural DNA nanotechnology: An evolving symbiotic relationship. *Biochemistry*, 42(24):7259–7269.

Sirringhaus, H., Sele, C. W., von Werne, T., and Ramsdale, C. (2006). Manufacturing of Organic Transistor Circuits by Solution-based Printing.

Stan, M., Franzon, P., Goldstein, S., Lach, J., and Ziegler, M. (2003). Molecular electronics: from devices and interconnect to circuits and architecture. *Proceedings of the IEEE*, 91(11):1940–1957.

Thompson, A. (1996). An Evolved Circuit, Intrinsic in Silicon, entwined with physics. *Proceedings of the First International Conference on Evolvable Systems: From Biology to Hardware*, 1259:390–405.

Trimberger, S. (1998). Scheduling designs into a time-multiplexed FPGA. *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 153–160.

Vinge, V. (1993). Technological Singularity. *VISION-21 Symposium sponsored by NASA Lewis Research Center and the Ohio Aerospace Institute, March*.

Waskiewicz, A., Groninger, J., Strahan, V., and Long, D. (1986). Burnout of power MOS transistors with heavy ions of Californium-252. *IEEE, DNA, Sandia National Laboratories, and NASA, 1986 Annual Conference on Nuclear and Space Radiation Effects, 23rd, Providence, RI, July 21-23, 1986) IEEE Transactions on Nuclear Science (ISSN 0018- 9499),*, 33(pt 1):1710–1713.

Wellekens, D. and Van Houdt, J. (2008). The future of flash memory: Is floating gate technology doomed to lose the race? *2008 international Conference on Integrated Circuit Design and Technology*, 189–194.

Winfree, E. (1998). Simulations of Computing by Self-Assembly. *Caltech CS Technical Report 1998.22*.

Winfree, E. (2003). DNA Computing by Self-Assembly. *The Bridge*, 33(4):31–38.

Winfree, E. and Bekbolatov, R. (2004). Proofreading tile sets: Error-correction for algorithmic self- assembly. *DNA Computing*, 9:126–144.

Winfree, E., Liu, F., Wenzler, L., and Seeman, N. (1998). Design and self-assembly of two-dimensional DNA crystals. *Nature*, 394(6693):539–544.

Wong, W. S., Daniel, J. H., Chabinyc, M. L., Arias, A. C., Ready, S. E., and Lujan, R. (2006). Thin-film Transistor Fabrication by Digital Lithography.

Wyatt, P. and Raffel, J. (1989). Restructurable VLSI-a demonstrated wafer-scale technology. *Wafer Scale Integration, 1989. Proceedings.,[1st] International Conference on*, pages 13–20.

Xilinx, Inc. (2006). Xilinx, Inc. http://www.xilinx.com accessed 10/31/2006.

Zeng, A., Lu, J., Rose, K., and Gutmann, R. (2005). First-order performance prediction of cache memory with wafer- level 3 D integration. *IEEE Design & Test of Computers*, 22(6):548–555.

Index