Chapter 1

OBTAINING QUADRILLION-TRANSISTOR LOGIC SYSTEMS DESPITE IMPERFECT MANUFACTURE, HARDWARE FAILURE, AND INCOMPLETE SYSTEM SPECIFICATION

Lisa J. K. Durbeck
Cell Matrix Corporation
Blacksburg, VA, USA
ld@cellmatrix.com

Nicholas J. Macias
Cell Matrix Corporation
Blacksburg, VA, USA
nmacias@cellmatrix.com

Abstract

New approaches to manufacturing low-level logic—switches, wires, gates—are under development that are stark departures from current techniques, and may drastically advance logic system manufacture. At some point in the future, possibly within 20 years, logic designers may have access to a billion times more switches than they do now. It is sometimes useful to allow larger milestones such as this to determine some of the directions of contemporary research. What questions must be answered so that we sooner and more gracefully reach this milestone at which logic systems contain a billion times more components? Some problems include how to design, implement, maintain, and control such large systems so that the increase in complexity yields a similar increase in performance. When logic systems contain 10^{17} switches or components, it will be prohibitively difficult or expensive to manufacture them perfectly. Also, the handling and correction of operating errors will consume a lot of system resources. We believe these tendencies can be minimized by the introduction of a low-cost redundancy so that, in essence, if one switch or transistor fails, the one next to it can take over for it. This reduces effective hardware size by a factor in exchange for a way both to use imperfect manufacturing techniques, and through similar means, maintain the system during its life cycle. It may also be possible to use similar basic principles for a more complex problem, designing a system that can catch and compensate for operating errors, but with low enough cost in time and resources to allow incorporation into all large systems.

We suggest that such a system will be a distributed, parallel system or mode of operation in which systems failure detection is a hierarchical set of increasingly simple, local tasks run while the system is running.

Work toward answering these questions appears to also yield some useful ways to approach a more general question, of constructing systems when their structure and function cannot be completely predetermined.

Keywords: integrated systems verification, integrated systems design, fault tolerance, fault isolation, fault handling, Cell Matrix, cellular automata, nanotechnology, electronics, transistor, Von Neumann, Drexler, CPU/memory architecture

Introduction

Hardware and logic designs have come a long way. The transistors used in a modern single-chip CPU are several hundred million times smaller than the original transistor built in 1947. If a contemporary CPU were built with the original transistor technology, it would take up a space of roughly one square kilometer. Current ways to produce logic designs pack many more transistors into hardware than their predecessors ten years ago, and ten to twenty years from now there may be ways to produce hardware devices with a billion times more transistors or switches. Note that such an increase in fabrication density does not lie on most current technology road maps, such as ITRS. This prediction is instead based on the expectation that researchers will uncover fundamentally different technologies that cause a sudden jump in device density. At the end of the curve following Moore's Law, we may find that process technology begins a completely new curve.

There has been and continues to be strong economic incentive for miniaturization of logic designs and electronics. Although for some products this has been used to simply reduce the footprint, designers have also been freed to create larger and more complex designs as transistor density has increased.

How complex will designs be with a billion times more capacity available? And what of the fact that some of the work to uncover replacements for the field effect transistor is being done in scientific disciplines in which three dimensional structures are not at all unusual– ten to twenty years from now there could be ways to produce three dimensional hardware for logic designs.

Technical breakthroughs over the next ten to twenty years could come gradually, but may instead exhibit sudden leaps in progress as problems are solved, discoveries are made. The path will depend on many variables: the nature and timing of future breakthroughs, whether they combine to form a complete production method, and how rapidly these new ideas are put into practice. In addition to having much larger switch counts and much smaller package sizes for logic designers to work with, there may be production means, and product parameters, very different from the ones in use today. There are many hard and interesting research questions at this junction that are appropriately addressed both in academia and industry. They include:

1 getting involved in the analysis of nascent switch-production methods to model how well they fit the engineering requirements of integrated circuits and what the outcomes could be from using a particular new method. Considerations include the number of switches per unit area or volume inside the device (density), total number of switches inside the device (volume), operating condition limitations, operating speeds, power requirements, production costs, and the reliability of production method and of the product during its lifecycle;

2 coming up with designs, and design tool capacity, for effective use of 10^{17} transistors or switches, such as designs that will scale up gracefully or even seamlessly as density increases, and ways to produce designs that readily lead to production of larger switch counts;

3 coming up with more powerful design and verification tools to handle logic designs with many orders of magnitude greater scope and complexity, which is a topic dealt with by Hsiao et. al in Chapter **??**;

4 coming up with more flexible processes for the product path, from definition of a new product's requirements, through logic design, test and verification, and implementation in hardware. Processes should be flexible enough to permit things like:

   (a) co-development of design, test and build, with none of these steps assumed to be fixed;

(b) development of design and verification that exploit a new fabrication method's strengths and minimize its weaknesses; and

(c) blurring of test and build into a more iterative process that takes the imperfect nature of a build process into account, rather than assuming that a perfect build is possible or normative while an imperfect build is unusable.

We have done some work related to these four areas of inquiry. The question we are interested in is what logic designs are suitable for billions to trillions of times larger and more complex designs, particularly if you remove the assumptions of perfect hardware and completely predetermined usage. We propose a two layer hierarchy, in order to decouple logic and manufacturing issues. Figure 1.1 gives some idea of this approach and its effect on the production of logic systems. For a given logic design X, we build a two-layer logic design, with the lower layer Y representing what is actually built in hardware, and the upper layer either representing the logic design X, or representing a means by which X can be constructed, or often a representation of a bit of both, such as a hardware library containing the components of X and a means to copy them and lay them out onto the hardware layer Y. The lower layer Y contains the logic design needed to either represent X in an efficient manner, or to construct X, after which it represents X in an efficient manner. This means we have direct control over the hardware layout of Y, but not X, which means we can control the placement of X's gates and wires on Y, but this is one step removed from the native hardware and its placement of transistors or switches and wires. In exchange, we gain the capacity to have parts of X defined, or redefined, at runtime, when information about the hardware's imperfections is known. Similarly, we can use this capacity to cause the layout of X to be affected by other information at any time during the system's useful life, and can use it to do things like optimize X or its implementation for the situation in which the product is used, such as the specific inputs it gets, use patterns, changes in use or inputs, environmental conditions, damage incurred throughout its life, etc.

The lower layer Y of this two-level design is a homogeneous structure with local-only interconnect, one that appears to be a good fit with the expected strengths and weaknesses of the revolutionary, post-field-effect transistor production methods under development. An illustration of Y's structure is provided in Figure 1.2. Layer Y's specification leads to hardware with every square or cubic millimeter packed with simple building blocks that can implement transistors/switches and wires, or small-scale-integration-sized gates, logic blocks, and wires. We call these building blocks logic cells, illustrated in Figures 1.3 and 1.4. Logic cells combine low level signals and produce outputs, and individual cells' functions can be combined to make more complex logic blocks and functions such as memory, state machines, multipliers, floating point units, and so on: any digital logic design. However, they also have this additional property of supporting non-static, partially-predetermined functions. We achieve this partly from the design of the logic cells, and partly from the functional directives we provide them. That is, we assign them not a static but a dynamic function, analogous to switch statements in programming languages (if X then do A, else B). Then we base their function on the inputs they receive, and organize groups of cells together to do useful work as dynamic circuits. The Y layer of hardware can be thought of as smart transistors and wires that can move around and change system function to suit new directives, and the new capability of dynamically changing function can be thought of as a system that can process and modify both data and its own logic, its own circuits, its own layout. The hardware layer Y is the physical and logical substrate upon which these traditional static and new dynamic logic designs exist. The logic design layer X is the set of current logic designs plus designs that take advantage of the ability to change the logic and its layout during its operation, for example, circuits that take in, process, and modify circuits.

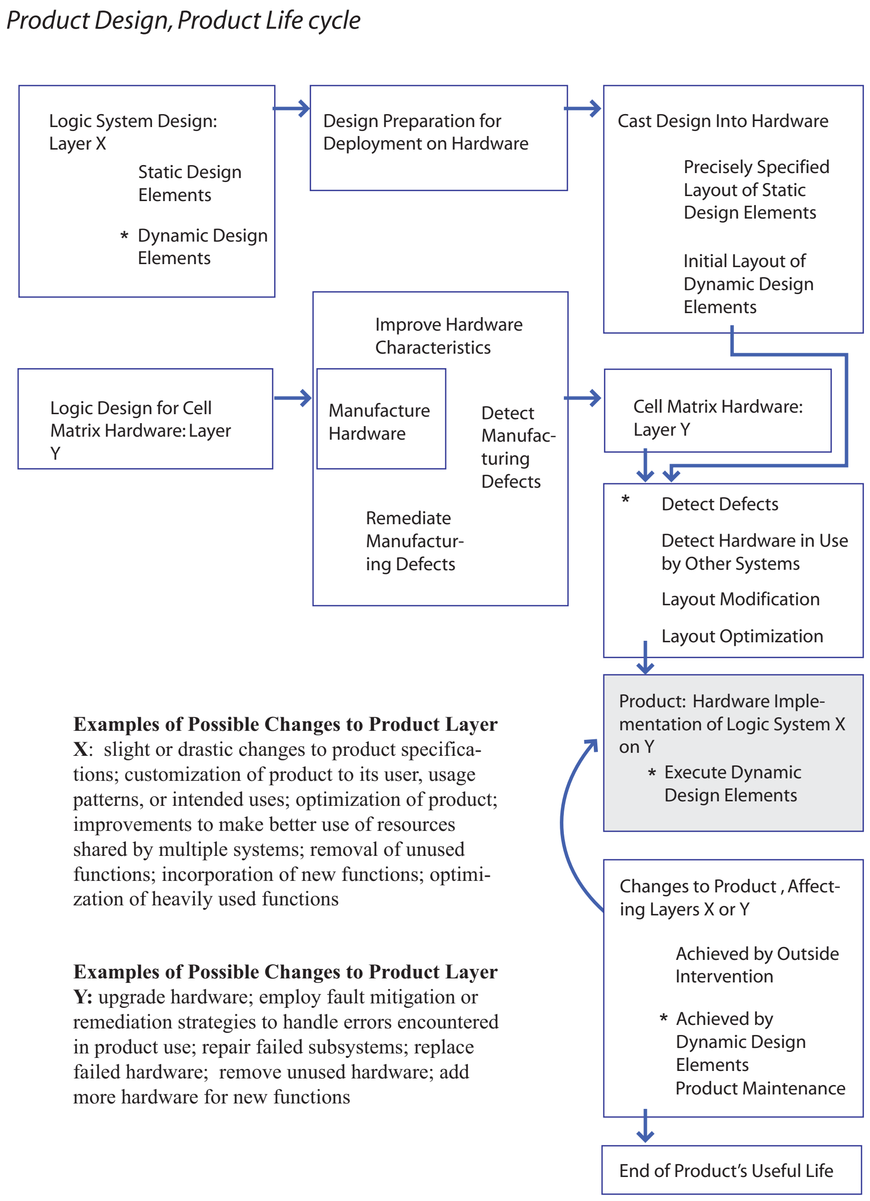
We have explored the use of this new capability to create systems that determine on their own how to best utilize their hardware resources, and to create systems that lay out perfect circuits on damaged hardware [2, 3, 5]; to create self-replicating and self-modifying systems [2, 3, 5, 10, 11]; to create an expanding counter that extends itself when it detects impending overflow conditions [11]; to create orderly copying of libraries of logic blocks onto Y hardware [10, 11]; to create mechanisms to test enhanced wires and transistors for faults and report them [2]; to create mechanisms to isolate faulty transistors and wires and demonstrate that they cannot affect the rest of the hardware [3]; to organize X,Y systems that efficiently perform computations via highly distributed, highly parallel functions [1, 12]. We have also ensured that traditional capabilities are well-served: it is easy to lay out a simple circuit onto Y hardware and can be done via straightforward engineering practice, for example, compilation of circuit schematics [14] or HDL [15]. We have fully defined the layer Y with a complete and simple specification [6, 8] and have initial, unsophisticated and fairly low-level ways to get X onto Y [14] and are working our way up to more sophisticated tools for the use of higher level representations of logic designs.

Our work to date can provide researchers with a convenient framework in which to develop research programs for these four areas of inquiry. In this chapter we describe and address one particular problem associated with so many orders of magnitude more switches per unit area/volume, which will give the reader a better idea of the problems that cause the need for new research in these four areas, and will provide a better idea of how to use our framework to approach them as well.

Figure 1.1 shows a high level view of how the production of a particular logic design might be performed using Cell Matrix hardware and the X,Y approach described here. The approach is similar to that used for reconfigurable devices today, with several important unique aspects that are marked in the figure by asterisks. Time proceeds roughly to the right in two parallel, independent tracks, and then down the right hand side of the diagram. The construction of the X and Y layers are performed independently, shown by the first two rows of events. The product is produced by the configuration of the Cell Matrix Hardware Layer Y according to the Logic System Design X, shown as steps proceeding down the right column of events. Because the implementation of X onto hardware is a post-manufacturing process, there is opportunity to perform useful functions in a post-manufacturing process labeled with a star, such as mapping and avoiding hardware defects, and modifying the design to accommodate other designs that are already on or will be put onto the hardware as well. Also, because the hardware is reconfigurable, the release of the product and sale to individual customers does not have to be the end of the product design and build cycle. Changes to the product or product line can be made after the product is released. Additionally, because this particular hardware layer Y can contain dynamic design elements, these product changes can be planned for and constructed ahead of time, put onto the hardware, and deployed in the field, triggered by an event during the product's lifecycle, and performed on the product hardware itself, with no external intervention needed. Examples of situations in which it could be beneficial to use this capability for the product to adapt and change are provided in the figure on the lower left.

1. Cell Matrix Overview

The Cell Matrix^{TM}is an architecture for a novel type of reconfigurable hardware system. Similar to an FPGA, the Cell Matrix is composed of a large number of simple reconfigurable elements (cells). Unlike most FPGAs though, there are essentially no internal structures besides the cells themselves. Moreover, each cell is connected to only a small set of neighboring cells. These two characteristics mean that the Cell Matrix


Product Design, Product Life cycle
Logic System Design: Layer X
Static Design Elements
* Dynamic Design Elements
Design Preparation for Deployment on Hardware
Cast Design Into Hardware
Precisely Specified Layout of Static Design Elements
Initial Layout of Dynamic Design Elements
Improve Hardware Characteristics
Manufacture Hardware
Detect Manufac-turing Defects
Remediate Manufactur-ing Defects
Logic Design for Cell Matrix Hardware: Layer Y
Cell Matrix Hardware: Layer Y
* Detect Defects
Detect Hardware in Use by Other Systems
Layout Modification
Layout Optimization
Product: Hardware Imple-mentation of Logic System X on Y
* Execute Dynamic Design Elements
Changes to Product , Affect-ing Layers X or Y
Achieved by Outside Intervention
* Achieved by Dynamic Design Elements
Product Maintenance
End of Product's Useful Life

Examples of Possible Changes to Product Layer X: slight or drastic changes to product specifications; customization of product to its user, usage patterns, or intended uses; optimization of product; improvements to make better use of resources shared by multiple systems; removal of unused functions; incorporation of new functions; optimization of heavily used functions

Examples of Possible Changes to Product Layer Y: upgrade hardware; employ fault mitigation or remediation strategies to handle errors encountered in product use; repair failed subsystems; replace failed hardware; remove unused hardware; add more hardware for new functions

Figure 1.1. A Way to Organize Production and Product Life Cycle for the X,Y Approach that uses Cell Matrix hardware. A high level view of how a logic design is turned into a product using this approach. The approach is similar to that used for reconfigurable devices today, with the unique aspects marked by asterisks.

architecture is inherently fault isolating: defects in a cell will generally have limited scope.

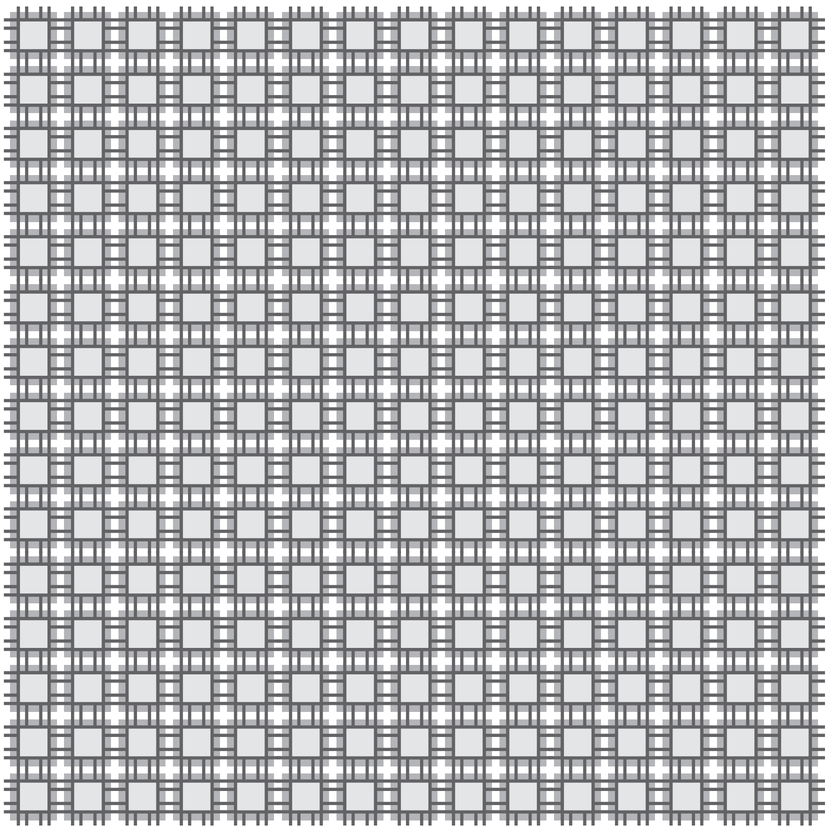


Figure 1.2. Hardware Layer Y definition. The hardware is a densely packed substrate containing a regular mesh of identical, very fine-grained SSI-scale processors that are connected in a local, nearest-neighbor scheme. The definition of the processors includes cases for connections with n local neighbors, 3, 4, 6, or other convenient local connection schemes; here, the processors are shown as 4-sided cells each connected to their four neighbors. The content of each of these processor cells is detailed in Figure 1.3.

The other essential feature is that, in contrast to an externally-controlled FPGA, the Cell Matrix is a self-configurable system. This means that the cells within the system are able to analyze and modify other cells, without intervention or guidance from outside the matrix. This is one key to efficiently managing the large amount of resources expected to be available in systems within the next decade. This autonomous, dis-

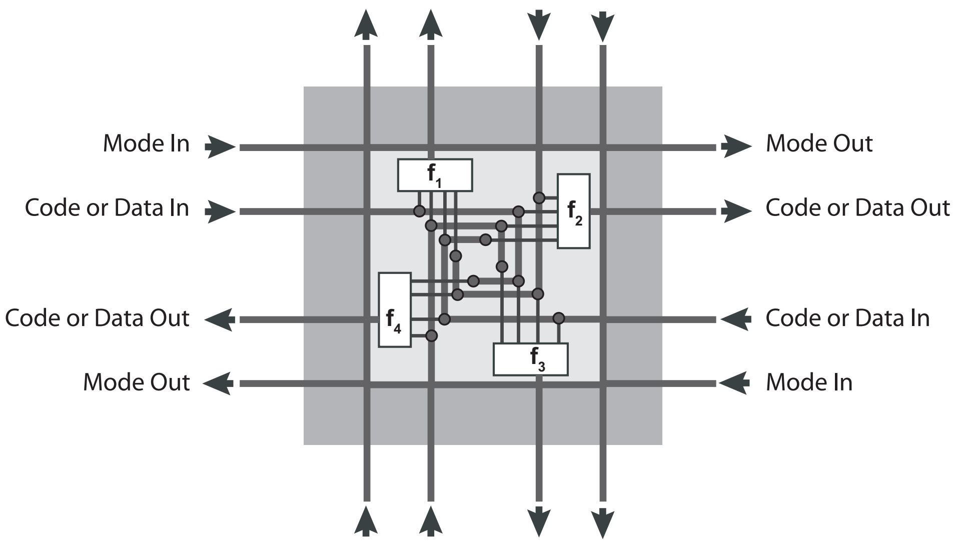
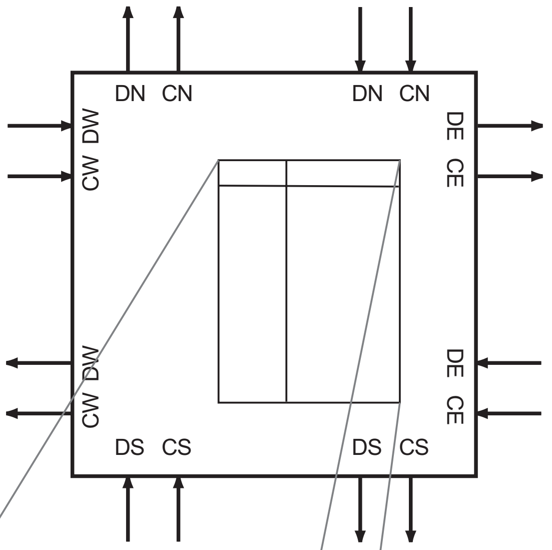


Figure 1.3. Structure of each logic processor cell in the hardware Layer Y from 1.2. A four-sided cell is shown. As shown by the input signal labels on the left of the figure, each side provides a data input to the cell, and receives a data output (which can also be used to output the cell's content, or code), as well as a Mode output that can be used to put the neighboring cell into a programmable control mode called C mode, during which the neighbor's content or code is changed. The input signals are combined to produce any logic function of four inputs and eight outputs, labeled f_4, that is delivered to the left side of the cell. The other three sides' outputs are other logic functions f_1 through f_3 that are based on any of the inputs to the cell. The functions f_1 through f_4 are settable: they are reset when the "Mode In" is set high, at which point the "Code or Data In" line is used to input a new code for the functions. The cell is completely symmetric in its function. A different view of the cell structure and function is provided in Figure 1.4.

tributed control is also key to managing run-time operational failures, since system behavior can be observed in many locations simultaneously.

The Cell Matrix architecture does not specify a particular topology or the system's cells. Cells may be three-sided, four-sided, six-sided,



D	D	D	D	C	C	C	C	D	D	D	D
N	S	W	E	N	S	W	E	N	S	W	E
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	1
0	0	1	1	0	0	0	0	0	0	0	1
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	1
0	1	1	1	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	0	1
1	0	1	0	0	0	0	0	0	0	0	1
1	0	1	1	0	0	0	0	0	0	0	1
1	1	0	0	0	0	0	0	0	0	0	1
1	1	0	1	0	0	0	0	0	0	0	1
1	1	1	0	0	0	0	0	0	0	0	1
1	1	1	1	0	0	0	0	0	0	0	1

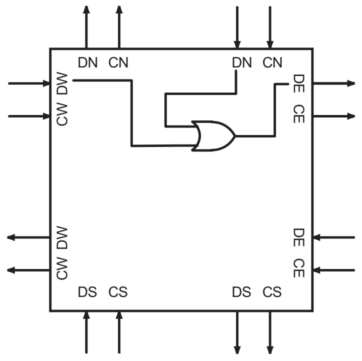
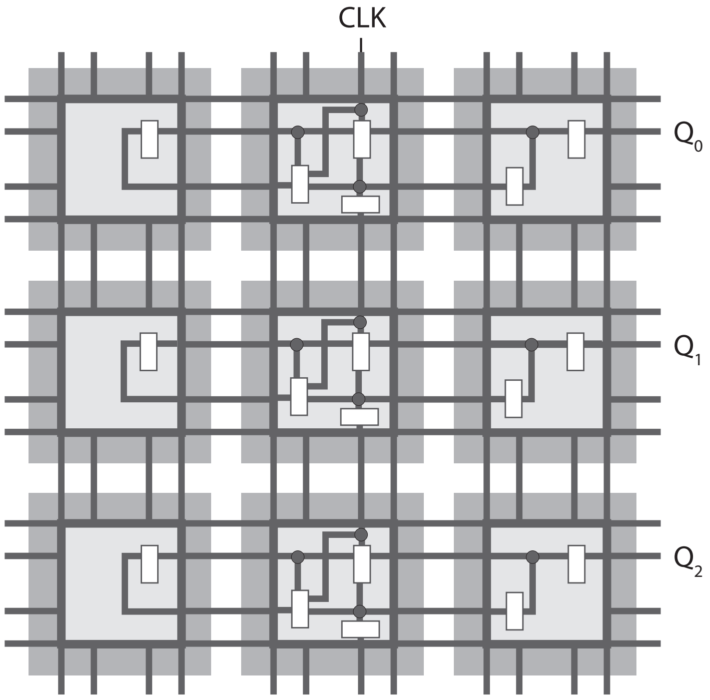


Figure 1.4. Logic Cell Structure and Function. The Signals in and out of the cell are shown, with the sides of the cells labeled as North, East, South , and West. The label "DN" at the top indicates the Data or Code signals sent in or out the North side of the cell, and "CN" indicates the Control Mode signals sent in or out of the North side of the cell. The lookup table that dictates cell function is shown and enlarged on the lower left. This particular lookup table configuration corresponds with the logical OR function shown on the lower right. A multi-cell lookup table configuration is shown in Figure 1.5.



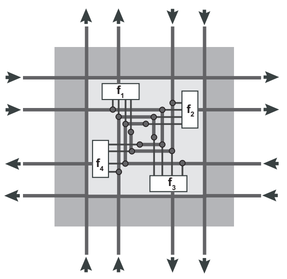


Figure 1.5. Six logic cells set up to function as a 3-bit counter. The same functional view from Figure 1.3 is shown in the inset at the bottom of the figure. Only the functions f_1 through f_4 that are used for this circuit are displayed in the upper 3×3 matrix of logic cells. A Clock signal CLK is provided at the top, and the bits Q are produced on the right. Many of the functions f_1 through f_4 are in this case used for simple wires to propagate signals; others in the center column of cells are used to perform boolean logic needed for the counter.

or any other number of sides greater than two (though two-sided cells have limited usefulness). Cells may be interconnected in two-dimensional or three-dimensional topologies - topologies greater than three dimensions are also possible. Moreover, the neighborhood defined for each cell can vary from one matrix to another. In practical terms though, most work to date has studied two-dimensional four-sided cells, and three-dimensional six-sided cells, in the expected square or cubic orientation.

Regardless of these specifics, cells and their arising matrix all operate along identical principals. Each cell has two inputs on each side, labeled D and C. Each cell has a corresponding set of outputs (D and C). Cells are interconnected according to the matrix-wide definition of a neighborhood, with inputs and outputs connected in the obvious fashion. Additionally, each cell contains an internal lookup table (LUT). The LUT maps every possible combination of D inputs to a set of C and D outputs.

Each cell within the matrix operates in one of two ways, depending on the *mode* in which the cell is operating. If a cell is in *D mode*, it continually samples its D inputs, looks up a set of outputs in its internal LUT, and sends those LUT values to its C and D outputs. Note that this happens continuously, without any clocking or synchronization.

If, instead, a cell is in *C Mode*, it samples its D inputs as specified by a system-wide clock, and loads the sampled bits into its internal LUT. Moreover, before a LUT bit is overwritten, it is sent to the cell's D outputs. C Mode is thus the mode in which a cell's LUT is read or written, while D Mode is the mode in which a cell is able to perform data processing functions via it's LUT.

Finally, a cell's mode is specified by its C inputs. If any C inputs are asserted, then the cell is in C Mode. Otherwise the cell is in D Mode. Three key consequences of this are:

1 the mode of any cell C1 can be specified by any of its neighbors C2 (since C2 has a C output connected to one of C1's C inputs);

2 the mode of a cell can change over time, since the value of its C inputs can change over time; and

3 a cell's mode is more or less independent of that of other cells - it is not a system-wide property, but a property of each cell.

The interaction of D Mode and C Mode cells thus allows cells within the matrix to read, modify and write other cells' LUTs. The LUTs can be processed as ordinary data, shared among cells as data, and then later used to configure cells, i.e., treated as code. This can be used to yield a number of powerful functions, including the testing of cell behavior, the creation of dynamic circuitry under the direction of the matrix itself, and the configuration of large numbers of cells in parallel.

Further technical details on the Cell Matrix can be found in [17].

2. Example of Future Problems: Lower Reliability

New production methods for logic designs will not necessarily have the benefit of utilizing past gains in reliability, since they may deviate greatly from past methods of production. Reliability is a big issue when considering a replacement for the existing process for creating hardware and for gradually miniaturizing switches. What is the error rate of the production method, and how many product defects does it produce? What specific types of errors and defects are encountered? How well does the product perform, and how long does it last? What specific types of error conditions and error states does it encounter during use, and are these error states unrecoverable? There may not be a single characterization of a production method with respect to these reliability questions, and the characterization will depend not only on the production method but on the product design as well: is the product robust against the types of errors typical for the production method? Operating failures are highly likely when using a relatively untested production method, and it will be important to gather empirical data on early products to be able to assess how well a new manufacturing method performs.

What the reader should be interested in is not just being on the receiving end of new hardware manufacturing methods, but also participating in their development, and their improvement. A new manufacturing process' expected density and volume are important, as are its expected cost and operating parameters such as speed, temperature, and power. However, reliability is also a big issue when considering a replacement for existing methods of manufacture. It seems unlikely that any radically different production method will have reliability even close to the current methods of producing field-effect transistors, because they will likely be too different to be able to incorporate the last five decades of improvements in transistor production. And yet they may have highly desirable traits in terms of volume or density, or operating parameters, and so we will want to be able to find a way to overcome their reliability problems enough to begin using them, and gradually improve them until they are reasonably reliable. Even if they eventually reach the reliability of current techniques, their reliability will forever be challenged by the tendency to utilize more switches in designs as density improves, what we are calling here increased volume. Therefore, even if the defect or operating error rates remained fixed, the number of defects or operating errors will continue to rise with volume. With volumes approaching a billion times today's volumes, operating errors will likely be commonplace in products as they are used.

Even with a manufacturing method that exhibits a small defect rate, such as a one in a million defect rate, logic designs with 10^{10} to 10^{17} transistors will be hard to manufacture perfectly, and, during their operation, will exhibit a mean time between failures that is markedly decreased, and reciprocally related to hardware volume, or system size. How are logic designers to handle this? Can designers continue to assume that the manufacturing process will produce perfect implementations of their designs that will also operate perfectly? Must logic designs be intimately aware of and tied to the physical constraints of the manufacturing method and its typical operating errors? Must each contemporary design be redone to increase its robustness against manufacturing defects and operating errors? A good goal for research in these areas would be to solve these reliability problems and yet allow logic designers to have to change as little as possible about what they do now.

With our two-layer approach there is opportunity to greatly minimize what logic designers have to know about the manufacturing method and its imperfections. This is because the view of the hardware is abstracted up one level above the physical layer, and is handled later in time, post-manufacture. There may be errors in the manufacture of the Y layer, but designs are laid onto the Y layer in a later step, which has the advantage of knowing what defects are present in the Y layer. Current research with our collaborators is intended to provide ways to handle operating errors that can exist within, underneath, alongside, or above the desired logic design X, using the dynamic capabilities of our architecture. The intention is that these capabilities can simply be evoked by the use of base-level logic blocks with extra capabilities for fault handling, dynamic system modification, and other needed functions. Although in some cases this will be an overly reductive approach, it is possible to present a view of the hardware as perfect to the logic designer, and delegate the handling of imperfections to a post-processing step performed on the design. This approach also supports the basic premise of managing complexity through encapsulation.

The focus of the discussion below will be on how two aspects of this approach–redundant hardware and dynamic functions–lead to ways to tolerate the reliability problems of larger, more complex systems built using immature production technology.

Manufacturing Defects

Manufacturing methods must be refined and improved, but perfection is a difficult goal to hold, particularly when the desire to continue to miniaturize switches persists in driving manufacturing onward to new challenges. There are at least five conceivable ways to handle defects in the construction of logic designs in hardware:

- discard any hardware that is not perfect;
- repair, or remove and replace, the individual defects;
- build redundancy into the hardware and a means to use only perfect resources within the hardware;
- use logic designs for Layer X that can function despite defects or runtime faults; or
- perfect the manufacturing technique so that it creates no defects.

The tact typically taken today is to place the burden largely on the manufacturing technique to provide perfect hardware, and to discard any hardware it constructs that is not perfect. We argue, however, that certainly during the development of revolutionary new fabrication techniques, and quite possibly long after their refinement, this will not be the most cost-effective option to choose. The other approaches above should also be considered, such as building redundancy into the hardware. Incorporating redundant copies of hardware components is used widely today to effect fault tolerant systems. This option is readily available for use in our approach, although it takes different forms for the different layers. Again, there are two layers to a logic design in our approach, the normal logic design X, and the lower level implementation layer Y. It is possible to incorporate redundancy into the logic systems design and implementation, as it is done today for systems onboard satellites and spaceships, via modifications to the logic design layer X. The lower layer Y can also be used quite effectively to safeguard products against manufacturing defects. The basic mechanisms offered above for safeguarding a logic system's perfect function against manufacturing defects are to either enlarge the system hardware to include redundant copies of resources, or to go in and repair, remove, or replace defects. Layer Y can achieve both of these models. How is does this is the next topic of discussion.

To define our use of terminology, we are using the general term errors to refer to any sort of nonoptimal function, and looking only at one cause of errors, hardware failures of some kind. We use the term manufacturing defect for those failures that are turned up during initial testing of the hardware, and operating errors for all other hardware failures that turn up later in the product life cycle. We note here that hardware failures' effects can be persistent or transitory, periodic or aperiodic, and can appear predictable or unpredictable in their response to testing.

Redundant Copies of Hardware

One benefit of our approach is that some measure of both fault tolerance and redundancy is automatically provided for all logic designs X by the nature of layer Y. Y contains low-cost redundancy already, because the logic cells themselves are resources that can be used to implement transistors, wires, flip flops, truth tables, gates, logic blocks, state machines, or any other digital circuit component. If one cell is bad, a design layout tool can use the one next to it. This is a great way to provide redundancy, because the system designer does not have to decide how much redundancy to put in ahead of time, or where exactly to focus the extra resources (three copies of this subsystem, four of that); instead, resources are pulled from a general pool, and used as cleverly as the design layout tool or diagnostic system is designed to utilize them. The problem is thus reduced to the question of how much larger to make the hardware than would be strictly needed for perfect manufacture, which could potentially be answered statistically, or via over engineering. This approach would then require an additional processing step in the design flow that modifies the placement of logic and wires using knowledge of hardware defects. A first version of such a tool for this X,Y approach was created by Dimitri Yatsenko [15]. We have also demonstrated the Y layer's capacity to make the necessary determinations on its own as to what hardware is good and where to put the gates and wires in a circuit definition it receives [2, 3, 5], which serves as an example of how to use dynamically interpreted, non-predetermined directives.

Nearly all circuits in use today cease to function properly when they incur any sort of damage. Any error in the creation of its transistors or wires is usually fatal to a circuit hardware implementation. Our cells are a much more fault tolerant circuit design because an error in the production of one cell is generally limited in effect to the immediate neighborhood of the cell. Because the X,Y Layered approach described here lays a logic design X out on Y hardware in a post-manufacturing step, we can attack this problem of manufacturing defects from a different angle, invoking several different ways to ensure that a defective piece of hardware is not intimately and irrevocably tied to the implementation of a critical piece of logic. Figure 1.1 indicates a number of functions that can be performed during the layout of the logic design onto the hardware in the righthand side box above the gray Product box.

No particular piece of hardware is necessary in layer Y, aside from the receipt of power to each logic cell. There are no differentiated cells, no buses, no external memories, no specialized structures. In fact there is no heterogeneity at all: the hardware is simply a densely packed surface or volume of logic cells connected only to their nearest neighbors. layer X can be put onto layer Y in a post-manufacturing step, at a point in time after Y has been tested for defects, and after an inventory of defect type and location is made. At that point in time, the layout of layer X can safely be determined so that it uses only good hardware. Or, if errors are encountered at runtime, there is support for dynamically re-allocating Y's hardware at runtime instead. This change to design flow insulates logic designs from hardware defects. It prevents logic system designers from having to change their current view that they will receive perfect implementations of their logic designs. In a later section on operating errors, we describe how error avoidance can be done at runtime, to deal with errors that arise in systems later, post production, from parts that start to fail, or from environmental effects.

SCANDISK for Logic Designs

One way to achieve this design flow process that can lay out perfect systems on top of an imperfect Y layer is to implement something analogous to what is used for memory, or disks, today, such as a SCANDISK type of process that checks each region of hardware and constructs a map of all bad regions, which is then used during writes to strictly prevent the copying of data into bad regions of hardware. The system that performs the scanning must itself be nondefective, and it should have the goal of marking as little extra hardware off-limits as is possible. We have constructed such a process for scanning the Y layer and reporting failed regions of hardware [2, 5]. The process has the capability of marking off $n \times m$ regions of cells as defective if any one of the cells fails any one of the tests it performs; most tests set the logic cell up to perform a specific function, then tests it with a set of input values to make sure it provides the expected outputs for the given inputs. The size of the region tested and marked is set to whatever is convenient, if testing is done by an external system proceeding from an edge of the hardware, and to a square region of 44×44 cells if testing is done by autonomous agents set up on the hardware, with most of that space required for replicating the testing apparatus and the ability to mark sectors to all locations where the testing is being done. Testing begins at an edge of a defined region, and scans the whole region using only already-tested hardware. Because of the distributed, local nature of signal processing in the Y layer's structure and function, we were also able to set up the test process so that it ran efficiently, testing many independent regions of hardware at the same time [3].

Isolation of Good Hardware from the Effects of Bad Hardware

If the intention is to use hardware despite the presence of manufacturing defects, then no defective hardware can be used in the layout of logic designs, and the defects present in the hardware must not be allowed to affect the logic design's function. How we approach this in the X,Y approach is that the layout of layer X on layer Y must be one that avoids placing any logic or wires within defective regions. We also explicitly prevent defective hardware from interfering with logic system function.

This preventive step could be done by either physically removing the defects from layer Y's hardware, or by logically removing them from layer Y's functioning. This effectively stops the spread of defects and guarantees that they will not alter the operation of logic design X in any way.

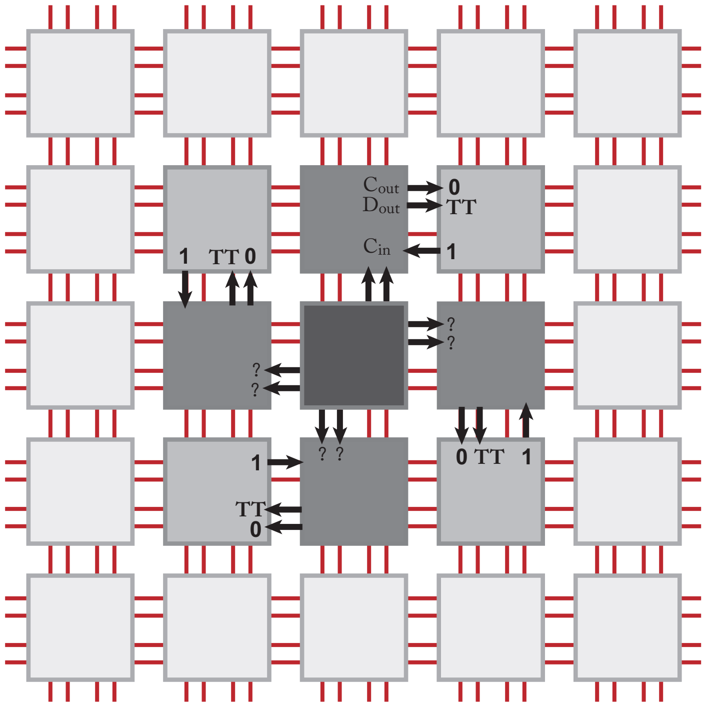
Layer Y has the desirable property that defects tend to remain localized near the source of the defect. This is largely due to the physical organization of layer Y, which is fine-grained cells connected only locally, to their nearest neighbor cells. We were able to validate this expectation by looking at failed cells' effects on silicon chips, and we have also studied failure modes by configuring reconfigurable chips with explicit errors in the definition of layer Y. Because there are no specialized structures within the hardware, and no distant connections, a defect is limited in its ability to spread. A cell C1 can fail, and for some types of failure modes, this cell's failure can send signals that affect the outputs of good neighboring cells C2, rendering them unusable. However, analysis of these scenarios suggests that there is only a very slight possibility that cells C2 will in turn affect their neighbors C3. The extent of the region affected by a specific defect will generally be one cell, and in some cases also its neighbors, and in a few of those cases also their neighbors, for a total of two connected levels outward from the defective cell.

This situation is a promising one, because it means that defects are naturally limited in their locality and effects. We go further, however, to ensure that defects do not affect good hardware. We do this by finding and then explicitly isolating faulty cells, as illustrated in Figure 1.6. They are explicitly isolated by setting the logic up in all neighboring cells to completely ignore any signals received from the offending side of the cell. This is described more fully in several sources [6, 5]; however, the effect is to construct a functional/logical wall around the defective region, by specifying the functions of the good cells at the boundary to ignore anything they receive from the defective region: it doesn't matter what they receive, they do not use it. This approach uses one set of cells surrounding a defective region to guarantee that no signals escape out of the wall around the defective region.

Figure 1.6 shows how this guard wall is erected. In the figure, it is assumed that analysis of each cell for defects has already been performed, and the center cell has been determined to be defective. Layout tools need to have knowledge of both the defects and the guard walls and are responsible for ensuring that they do not attempt to lay down a part in this region: if they try, they will fail, because the guard wall is already in place and irrevocable. They may use only the unutilized resources and sides within the outermost level of the guard wall. The defective cell is logically isolated from the functioning of the rest of the matrix by ensuring that its outputs will be explicitly ignored. This is done by putting its immediate neighbors, the plus shape of dark gray cells around the center, permanently into C mode, which causes them to send out low signals on the rest of their output lines no matter what signals they receive. They are put into C mode by the light gray cells, as indicated by the 1 on the C_in lines of the dark gray cells. This strategy completely contains the signals from the defective center cell. It uses up the four neighboring cells completely, and the edge of one of each neighbor one level further out in the adjacency that is used to send the C mode signal. The smaller figure below it shows that the guard wall grows compactly around the defect with a two-cell defective region.

Detection of Bad Hardware

Before defects can be examined and explicitly walled off, they must be found and pinpointed as precisely as possible. Because cells have the ability to exchange data with other cells, as well as the ability to change a neighboring cell's function, it is possible for one cell to perform a series of tests on a neighboring cell. For example, a cell can be configured to always output 0, and by then verifying that the output is zero, one can confirm the output is not stuck-at-1. Similarly, a cell can be configured to always output 1, to detect a stuck-at-0 fault in the output. By configuring a cell as a wire that outputs its input, one can partially verify the cell's configuration mechanism. Configuring a cell as an inverter allows one to check for a short between the input and the output of the cell, as well as to further test the configuration mechanism of the cell. More complex test patterns can be used to further exercise the cell-under-test. Using these basic concepts, we have developed ways to test for defects at a granularity of one cell, and the passage or failure of one test by that cell [2].



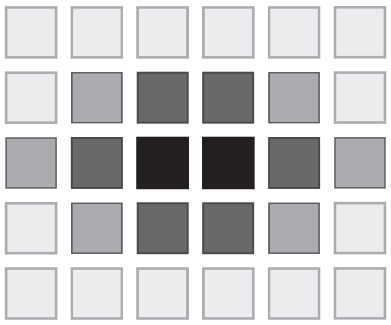


Figure 1.6. The Construction of a Guard Wall, and inset example of a guard wall for a 2-cell-wide defect. Black cells are defective, dark gray cells are used to isolate the defective cells' signals, and light gray cells are used to orchestrate this isolation.

Most defects can be detected by testing cell function, and extensive testing, while time-consuming, can be organized as a distributed, parallel set of local processes, so that many cells can be tested at each iteration of a testing regime (ideally an increasing number at each iteration to reduce the order of magnitude). We have constructed a distributed, parallel, order $\sqrt{n}$ means to access logic cells using only already-verified hardware to do so, and to supply tests of their functioning, and reporting of any failures [2]. These test patterns can then be customized to the expected defect types for a specific hardware manufacturing process.

Operating Errors

Even if a component of a hardware design is perfectly constructed, its inherent nature may cause it to suffer occasional lapses, intermittent failures, and at some point, permanent failure. There will be a certain probability of these operating errors occurring, and at some point in the scaling up of system sizes, systems will contain enough components that the summed probabilities of component failures lead to a rate of errors high enough that errors are completely common while a system is operating. The gravity of this situation and potential remediations are dealt with in Andre Dehon's chapter **??** . Operating errors are an example of the kinds of things for which a solution has to be provided at runtime, during the working lifespan of a product. Solutions generally cannot be preorchestrated or preordained, and they benefit greatly from the capacity of our X,Y approach to put higher level functions onto the X layer that are dynamically interpreted at runtime. This is analogous to telling someone what you want them to achieve, but not telling them explicitly how to achieve it.

As an example of how the our approach can be used to handle operational errors, we have implemented an autonomous, self-repairing system [5]. This system implements another layer S in addition to X,Y. This new layer is responsible for the following actions:

- detecting errors in the Y layer;
- isolating errors in the Y layer;
- populating the new S layer; and configuring the S layer so as to implement the desired target circuit from the X layer.

In the system we designed, one could define a target circuit, and layer S would implement that circuit automatically, without external guidance. This means the circuit designer does not need to know which parts of the hardware layer Y are defective, and does not need to participate in the layout of the target circuit on Y. Because of this, if a system failure is detected during operation, the system can automatically rebuild itself, working around the new defective areas.

This is, in a way, nothing more than an automatic place-and-route system, but with a few key differences:

- the place and route is being run in the same hardware, in a parallel, distributed fashion;
- the algorithm can be implemented on hardware containing defects;
- the hardware first self-organizes to create the circuitry for running the place and route; and
- the same hardware that implements the place and route algorithm will eventually implement the final target circuit.

Thus, by having layer S, the system no longer has to find, isolate and work around defects for **any** application. Instead, it solves these problems for a particular problem, that of the place and route controller. Once that controller has been implemented, **it** is then responsible for implementing the final target circuit. Since the place and route system is now aware of the location of faults in the hardware, it is a relatively simple task to avoid them in implementing the target circuit.

Repairing, Removing, or Replacing Defective Hardware

Another approach to manufacturing defects is not to accept them and handle them, but to go back in after manufacture and testing, and physically remove or replace them, or to effect repairs to them. This second defect-handling process could be done by the same process that built the hardware, by a sort of “try again” model, or by another process that specializes in removal and repairs, or by a combination thereof.

Because of the physical organization of the Y layer, it is amenable to either the replacement or removal of hardware. Missing hardware is no different from unused hardware, except in that it can never be called upon for use. Our layouts of layer X designs often have empty space within them now, to make them easier to view, or to add to or amend later. Also, the size of the hardware does not matter to the Y layer. It can be arbitrarily big in its dimensions: on the Y layer there are no presumptions or structures dependent on size, such as data buses and address spaces. And the hardware can be grown or shrunk at a later date. We have demonstrated the ability to make and use a larger piece of Y hardware by simply joining two pieces of Y hardware, by a simple physical process of simply joining cells’ neighbor wires along the edges of two pieces of hardware. Removed hardware would simply register as defective hardware via the detection mechanisms described above, which can be combined with layout procedures in a manner similar to that for defects, so that the layout of layer X does not include the use of any missing hardware or holes in the hardware.

The checking of repairs can also be handled by the testing and defect handling mechanisms described above, such as the analogue to SCANDISK. If the repairs are good enough, the cell passes all tests and is treated like any other good hardware; if they are insufficient, the cell gets treated appropriately as a defective cell. Iterating between repair and this process would be a good way to determine when the repair job has succeeded: the cell or region would no longer appear on the defect list.

Efforts to perfect the manufacturing technique, even if not met with complete success, are also desirable because they result in the highest density and volume for a fixed manufacturing process. This effort may also help to reduce operating errors. The construction of a simpler target is often easier to perfect. Accordingly, the construction of the Y layer is easier to perfect because it is simply a task of repeatedly constructing the same, small design for the logic cell, and repeating the same interconnection pattern throughout.

3. Summary, Conclusions

Revolutionary, not evolutionary, new manufacturing processes are a hot research topic. With the capacity to vastly increase the number of switches designers can use in logic designs, and with the likelihood that these new manufacturing processes will be too different from the current techniques to borrow many advancements from them, their introduction may well usher in an age where reliability cannot be guaranteed by the manufacturer and must be handled upstream in the design flow as well, and, to a much larger degree than is true today, downstream in the product also. Reliability must be directly addressed, on a per-design basis with logical/physical co-design where necessary and appropriate, and with a broader brushstroke wherever possible.

The broader brushstroke we advocate upstream, for design systems, is to address reliability using a two-layer, multi-step layout process we dub X,Y, where a convenient, tightly packed hardware design Y is built with low-cost, low-level redundancy. Then the hardware is tested, and then used to implement specific logic designs built up from SSI level logic components, wires, or state machines. This approach is much less dense than would be a direct hardware representation of the logic design X. However, it makes it possible to routinely use imperfectly manufactured hardware, which means that it is straightforward to scale hardware up to much larger designs without paying a large penalty in manufacturing yields, significantly increasing applications while simultaneously lowering costs. And it also means that a new manufacturing process can be put into production sooner, before it is perfected, particularly for small logic designs that can fit on the potentially small amount of good hardware in a device that early versions of a process will produce. Layout of a design onto faulty hardware would most likely need to be done by an automated process, since each piece of hardware will have different defects. For one-offs or layouts onto a specific piece of hardware, however, design tools could build fault locations into the design checks and simply disallow the placement of logic on particular regions of the hardware space. This capability could serve in a facility for diagnosing and then repairing customers' systems.

The manufacturing process must of course be improved as much as possible, and its unreliability duly minimized, but the design system will likely be left with something still imperfect, and must take it from there. Accommodations, such as these suggested here, should be developed so that hardware can be used that is imperfect, but has acceptable flaws or reliability problems. A manufacturer could use each piece of Y layer hardware it produces for whichever of the logic designs it is producing, whichever one automatic tools can find a way to fit.

Downstream, reliability hits due to operating errors will plague products containing massive numbers of switches. There are ways that have been developed to handle this within logic designs themselves such as subsystem redundancy and voting. Again we advocate using a broader brushstroke wherever possible, rather than a per-design fault handling strategy. The broader brushstroke we advocate downstream for operating errors is efficiently self-analyzing, efficiently self-modifying systems, using the dynamic, local processing capabilities of layer Y: investing designs with the ability to test their higher and lower level systems for operating errors, and make repairs, remove and replace subsystems by using hardware libraries and a means to create new copies elsewhere on the hardware, or in general by shifting and moving their logic to avoid using failing hardware. This is a wide area of research and there are likely different ideal ways to do this for different kinds of applications. The main benefit to this approach is that rather than trying to predict all failure modes or failure situations and invest all systems with a game plan for them, each circumstance is handled if and when it arises, by a more general approach, and the hardware is eventually fine-tuned to its own quirkiness, usage patterns, and environment. The vagaries of intermittent failures do not lend themselves to a static solution, but rather a solution as dynamic as the problem itself.

References

[1] Durbeck L and Macias N 2001 The Cell Matrix: an architecture for nanocomputing. Nanotechnology vol 12 pp 217-30 (Bristol, Philadelphia: Institute of Physics Publishing)

[2] Durbeck L and Macias N 2002 Defect-tolerant, fine-grained parallel testing of a Cell Matrix Proc. SPIE ITCom 2002 Series 4867 ed J Schewel, P James-Roxby, H Schmit and J McHenry pp 71-85

[3] Macias N and Durbeck L 2002 Self-Assembling Circuits with Autonomous Fault Handling Proc. The 2002 NASA/DOD Conference on Evolvable Hardware ed A Stoica, J Lohn, R Katz, D Keymeulen and R Salem Zebulum pp 46-55

[4] Redl F X, Cho K-S, Murray C B and O'Brien S 2003 Three-dimensional Binary Superlattices of Magnetic Nanocrystals and Semiconductor Quantum Dots. Nature, 423, pp. 968 - 971 (26 June 2003).

[5] Macias N and Durbeck L 2004 Adaptive methods for growing electronic circuits on an imperfect synthetic matrix Biosystems Volume 73 Issue 3 (Elsevier ireland Ltd.), Pages 173-204, March 2004.

[6] Macias N, Raju M and Henry L 1998 Self-Reconfigurable Parallel Processor Made From Regularly-Connected Self-Dual Code/Data Processing Cells. US Patent 5,886,537

[7] Culbertson B, Amerson R, Carter R, Kuekes P and Snider G, Defect Tolerance on the Teramac Custom Computer, Proceedings of the 1997 IEEE Symposium on FPGA's for Custom Computing Machines, pp. 116-123.

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

[9] Drexler K E 1992 Nanosystems: molecular machinery, manufacturing, and computation. Wiley Interscience.

[10] Macias N 1999 Ring Around the PIG: A Parallel GA with Only Local Interactions Coupled with a Self-Reconfigurable Hardware Platform to Implement an O(1) Evolutionary Cycle for EHW. Proc. 1999 Congress on Evolutionary Computation pp 1067-75

[11] Macias N 1999 The PIG Paradigm: The Design and Use of a Massively Parallel Fine Grained Self-Reconfigurable Infinitely Scalable Architecture. Proc. The First NASA/DOD Workshop on Evolvable Hardware ed A Stoica, D Keymeulen and J Lohn pp 175-80

[12] http://cellmatrix.com/entryway/products/pub/ForesightPoster.pdf

[13] von Neumann J 1966 Theory of Self-Reproducing Automata. University of Illinois Press, Urbana, Illinois.

[14] http://cellmatrix.com/entryway/products/software/layoutEditor.html

[15] http://cellmatrix.com/entryway/products/pub/yatsenko2003.pdf

[16] http://www.xilinx.com

[17] http://www.cellmatrix.com