Self-Modifying Circuitry for Efficient, Defect-Tolerant Handling of Trillion-Element Reconfigurable Devices

Nicholas J. Macias

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering

Peter M. Athanas, Chair

Cameron D. Patterson

Tom L. Martin

A. Lynn Abbott

Alex K. Jones

April 2011

Blacksburg, Virginia

Keywords: reconfigurable, self-configurable, self-modifying, FPGA, parallelism, fault handling, non-dualism, Avogadro machine

Self-Modifying Circuitry for Efficient, Defect-Tolerant Handling of Trillion-Element Reconfigurable Devices

Nicholas J. Macias

(ABSTRACT)

As VLSI circuits continue to have more and more transistors over time, the question of not only how to use, but how to manage the complexity of so many transistors becomes increasingly important. Four hypothesis are given for the design of a system that scales-up as transistors continue to shrink. An architecture is presented that satisfies these hypothesis, and the motivation behind the hypothesis is further explained. The use of this architecture's unique features to implement an efficient, defect-tolerant parallel bootstrap system is discussed. A detailed methodology for implementing this system *in vivo* is described. A sample problem — simulation of heat flow — is presented, and its solution using the proposed architecture is described in detail. A comparison is made between the proposed architecture and a set of contemporary architectures, and the former is shown to have desirable performance in a number of areas. Conclusion are given, and plans for future work are presented.

Dedication

To everyone who's ever lived, loved or dreamed outside the lines, this is humbly dedicated.

Acknowledgments

The author wishes to thank Peter Athanas for his guidance, wisdom, and patient review of this work and manuscript. The author is also indebted to the other committee members — Cameron Patterson, Tom Martin, Lynn Abbott and Alex Jones — for their participation in the various stages leading up to and including the final defense.

Contents

List of Figures

Chapter 1

Introduction

"Size matters not" [**?**]. So proclaims Yoda, the mythical Jedi Knight from the Star Wars series. Admittedly, that utterance was made a long time ago, in a galaxy far from Silicon Valley. But in 2011 in Silicon Valley, size *definitely* matters. The semiconductor industry is absolutely obsessed with the size of transistors. For several decades, devices have gotten smaller and smaller. This trend is not simply some side-effect, but is one of the primary driving forces behind innovation in the industry. It is so ingrained in the collective psyche that an *observed* trend — the doubling of the number of transistors per integrated circuit every two years — has been dubbed a "law"[**?**]. Continued reduction in transistor size forms a key part of the semiconductor industry's ITRS (International Technology Roadmap for Semiconductors) [**?**] — the supposed guide of *the* path that must be followed for future prosperity. Foundries are often identified by a single characteristic of their output: the gatewidth of manufactured transistors. Mention of a new foundry start with a smaller feature

size in a corporation's 10-K filing is certain to fuel investor confidence for another quarter. By multiple measures, the semiconductor industry is seriously invested in maintaining this trend of ever-decreasing transistor size, and with good reason: this continued reduction in transistor size has helped foster tremendous growth in the industry, and in industries that rely on semiconductor technology.

While transistors have continually shrunk in size over the past several decades, the integrated circuits containing those transistors have generally stayed the same size. This leads to the most noticeable effect of this ongoing reduction in transistor size: an increase in the number of transistors per fabricated device — again, Gordon Moore's observation. In the area of CPU design, this continual increase in transistor count has manifested itself in several different architectural aspects, including wider datapaths, more-advanced floating point capability, larger on-chip caches, and the inclusion of advanced architectural features such as branch prediction. This ongoing increase in the power of computing devices has influenced almost all aspects of modern life, including areas such as communications, networking, transportation, energy and medicine. By most measures, this would certainly be considered a highly-desirable state of affairs.

But there is a storm coming — a dark cloud nestled inside the silver lining of lithographic innovation. Its arrival is inevitable, for it is directly tied to the very thing the industry craves: the ever-increasing density of fabricated devices. This increase in density and transistor count is certainly advantageous in many ways. However, as the number of transistors increases, several technical problems grow more severe.

- The design time for systems utilizing a large number of transistors grows much more than linearly with the device count.

- As designs grow more-complex, design cost is also ever-increasing, including the overhead of managing ever-larger design teams. Other NRE (non-recurring engineering) charges also continue to increase.

- The cost of building a new chip foundry is roughly doubling every 2–3 years. This is Moore's Second Law [**?**].

- As gate-widths continue to shrink (along with other fabricated features such as metal lines), the defect rate in fabricated devices continues to rise, with yields dropping accordingly.

- As switching voltages drop and clock frequencies rise, the likelihood of upsets in transistor outputs increases, and the mean time between failures (MTBF) drops.

- The number of I/O pins per packaged device has grown very modestly compared to the number of transistors. Thus the number of transistors per I/O pin is growing rapidly. Put differently, the I/O bandwidth available per transistor is continuing to shrink.

- The combination of faster switching times, smaller wires with higher resistance, and increased transistor density means that heat generation within a chip continues to increase.

- As clock frequencies increase and clock lines are distributed to more and more transistors, clock skew becomes a bigger problem.

These are some of the challenges currently faced by designers and manufacturers of integrated circuits. Each time feature sizes drop, these problems grow a little more severe. So far though, these reductions in feature size have occurred relatively infrequently — every couple of years. This gives the industry time to make incremental improvements, for example, in the design process, in manufacturing capabilities, and in ways to mitigate or handle faults and defects. In this sense, the fact that development is not occurring more-quickly is actually advantageous to the industry.

This situation may soon change though. While there is concern that Moore's law may one day end, in the sense that progress may falter, there is another possible scenario: the end of Moore's law because of a sudden, dramatic *increase* in device density. Such a hypothetical event has sometimes been called *The Singularity* [**?**]. This dramatic increase may come about from a combination of several factors, including:

- the development of a fundamentally new, highly-scalable switching technology to replace the transistor [**?**];
- developments in quantum computing [**?**];
- development of fabrication techniques that allow physically-larger devices to be manufactured (i.e., wafer-scale integration [**?**]);

- the advent of our ability to perform true atomic-scale manipulation of matter (as proposed by Richard Feynman in 1959) [**?**]); and
- a transition from fundamentally flat fabrication to the creation of full three-dimensional structures (as opposed to two-dimensional structures with a small amount of height in the Z axis) [**?**].

At such a point, the industry for creating computing circuits may resemble chemistry more than electronics.

In chemistry, a common unit of measure is a *mole*, defined as "the amount of substance of a system that contains as many *elementary entities* (e.g. atoms, molecules, ions, electrons) as there are atoms in 12 g of carbon-12 (12C)" [**?**]. At the point when circuits are built by placing individual atoms, it may make sense to discuss a "mole of switches," i.e., a circuit containing approximately 6.02×10^{23} transistors (or transistor-like switches). Such a system would be over *fourteen orders of magnitude* more complex than today's most-advanced computing devices.

Supposing such a singularity does occur — and it's certainly a good possibility, given the growing interest in such breakthroughs, and the development of facilities and programs to support such research, e.g. Ray Kurzweil's "Singularity University" [**?**] — what will the chip-design landscape look like? Considering the above complicating-factors, given an increase in complexity of more than a trillion times, one might predict the following trends:

- design time and design cost will each increase to the point of being infeasible;

- defect rates will make fabrication of perfect devices extremely difficult;
- the ratio of transistors to I/O pins will become huge, leading to a continuation of the long-standing problem of I/O bandwidth; and
- total system-wide heat generation will increase dramatically [**?**, **?**].

One might also expect a number of other formidable technical challenges, including the development of fundamentally new design tools.

The real challenge here is that the present way of doing things — the basic architecture of systems, the way those systems are designed, the way they are manufactured, and how they behave at runtime — simply does not scale very well. With much work, the industry can almost keep pace with a bi-annual doubling in transistor count. But it's unlikely that present methodologies will scale well in response to something like a sudden, one hundred trillion times increase in switch density. Part of the difficulty lies in the basic assumptions about how to design next-generation devices:

- an expectation of fabrication to produce perfect, defect-free chips;
- a hope to ramp up clock speed despite its effect on heat generation and problematic clock skew;
- the use of packages with a relatively small number of pins, and reliance on those few pins to monitor and control an ever-increasing number of transistors inside the fabricated chips;

- the use of a basic system architecture — the Von Neumann stored-program CPU/memory architecture — that is several decades old;
- the building of new foundries whenever physicists find a way to do fabrication at a smaller scale; and
- a practice of completely re-designing the systems being built each time the device density increases.

A different approach is needed — a different way of doing things:

- The design of systems needs to be changed, to something more-scalable, something that does not need to be re-designed when there are more switches available. This probably means something composed of many homogeneous elements, where it's possible to simply "add more elements" and end up with a larger, more-powerful system. It also suggests a fairly simple interconnection scheme among those elements.
- The *manner* in which systems are designed needs to change. Having teams of people designing systems by hand, and using single-CPU or modestly-parallel workstations to compile those designs, is simply not going to be practical. A better approach might be to create systems that are able to *design themselves*, perhaps using their own massive parallelism to effect more-efficient place-and-route, etc.
- The ever-decreasing bandwidth of I/O subsystems needs to be dealt with, by finding ways to manage complex systems *from within* those systems themselves, instead of

trying to control them through a relatively small number of macroscopic I/O pins.

- The way in which speed is extracted needs to be changed — achieving speedups through massive parallelism vs. ever-increasing clock frequency.

- The way in which defects and runtime faults are viewed needs to change — accepting them as an inevitability and learning how to function in their presence, instead of assuming they can be ignored.

So how can system architecture, complex circuit design, and run-time behavior be re-thought given the above considerations? The proposed answers discussed herein are based on four observations/hypotheses about industry trends.

1.1 Hypothesis about future computing

Hypothesis 1: Industry trends point to a continuing shift towards *extremely* large reconfigurable devices.

Two primary trends in the industry are working fundamentally against each other. This first trend — ever-increasing density and transistor count — suggests that devices will continue to grow more complex over time. The second trend — increasing costs associated with this growth — suggests that custom ASIC is becoming unaffordable to all but the largest entities (those that can amortize NRE costs across a huge volume). The combination of these trends thus suggests a continuing move to *reconfigurable devices*: hardware that has a fixed design,

but can be customized post-manufacture by the user for their specific needs. This is the first hypothesis of this dissertation.

Hypothesis 2: As the number of transistors in reconfigurable devices continues to increase, it will become increasingly useful if those transistors can be monitored and manipulated from inside the device itself.

This is based on consideration of additional trends which further complicate the long-term picture. As switching voltages drop, the likelihood of glitches — momentary changes in the logical value of a signal — increases. As the physical size of transistors shrinks, the probability of a manufacturing defect — a short or open circuit, or a misalignment between masks — also increases, as does the probability of undesirable run-time phenomena such as metal migration. The increased likelihood of these and other failures within the reconfigurable device suggests a growing need for constant monitoring of system behavior. Normally, this monitoring can be done using external circuitry to access different points inside the system. Such monitoring can make extensive use of I/O resources to provide sufficient addressability for probing numerous points inside the system. However, while the number of transistors within a device has grown rapidly, the number of I/O pins available on a package has grown much more slowly. Rent's Rule [**?**] relates the number of pins P on a device to the number of gates G contained in the device as $P = tG^q$, where t and q are constants, and $0 < q < 1$. Re-writing this equation, the ratio of gates to pins $G/P = (1/t)G^{1-q}$.

This suggests that continued reliance on external systems to analyze and manage the transistors within a device will lead to increasingly long delays as more and more transistors

are accessed from a comparatively small number of I/O lines. To circumvent this problem of limited I/O, one can consider a system design where the monitoring and control of the system's transistors is done, not by external systems, but *from inside the system itself.*

Hypothesis 3: Having the internal management circuitry itself be reconfigurable increases the flexibility and general usefulness of the system. This is because, as the number of reconfigurable elements grows *larger*, the circuitry for managing those elements is also going to grow, and probably become more-complex. And in the same way that an FPGA is sometimes a useful alternative to a fixed custom ASICs, a reconfigurable management system may be more useful than a fixed one. This is simply because decisions about exactly how that system should be designed can be made post-manufacture, and can be changed based on how the reconfigurable elements are going to be configured and used.

Hypothesis 4: If the reconfiguration of management circuitry is indistinguishable from the reconfiguration of target circuits, then you eliminate hierarchy, or in some sense "close the loop" between the top-level and the bottom-level of the hierarchy. This is more subtle than the other hypothesis, and will be covered in more detail in the next chapter. The basic idea is that, if the control/management layer is itself reconfigurable, then what will control *that* layer's configuration? If R is the set of reconfigurable elements, and C_1 is the control layer for R, and C_1 is reconfigurable, the there must be a control later C_2 for controlling C_1. And while C_2 can be fixed/hardwired, the issue is, as R grows, the size and complexity of C_1 will also grow, and thus so will C_2. If R gets large enough, C_2's size and complexity may become large enough that having C_2

itself be reconfigurable may be desirable, leading to the inclusion of *yet another control layer* C_3, and so on. One way to break this chain of ever-deeper layering of control systems is to simply allow C_1 to configure itself, i.e., to play the role of C_2, C_3, and so on. An even simpler solution is *to allow R to be its own stack of control layers* $\{C_n\}$. This is the essence of Hypothesis 4: to have the set of reconfigurable elements and the set of *reconfiguring* elements be indistinguishable from each other, because they are exactly the same set of elements.

1.2 Prior Art

The basic architecture discussed herein has been previously explored, under the name "Cell Matrix" [**?**, **?**, **?**]. While the Cell Matrix is a useful instance of a particular self-configurable architecture, the concepts developed in this work are meant to be largely independent of any particular self-configurable, hierarchy-free architecture. The present work, however, does extend prior Cell Matrix research in a number of ways, including the development of new tools, new circuits and sequences, and new methodologies for internally-directed creation of circuitry.

1.3 Bio-Inspiration

Much of this work, including the background work in the Cell Matrix, derives from ideas found in biology. General concepts such as using the parallelism of a large number of slow,

simple processors to perform powerful computations (vs. relying on a single complesx, extremely fast CPU) have obvious analogues in natural systems. Similarly, the idea of reducing critical areas in a system, i.e., minimizing the number of points where a single failure will render the entire system inoperable, is found in many biological systems. While the present work does not seek to *mimic* nature, it does draw from examples in nature for motivation and ideas for new directions in which to explore.

Moreover, there are directions in which this work could be taken that derive more-directly from biology, such as in implementation of neural networks, or the development of embryological circuitry. While such directions would most-likely prove to be fruitful, the present work is not focused on such applications, but rather on an analysis of the underlying concepts of self-configuration, and its usefulness in support of extremely-large reconfigurable systems.

1.4 Scope

A number of tools have been developed in the course of this work, but it is important to note that these tools are exploratory in nature, and are primarily intended as a means to an end, namely a deeper understanding of certain concepts related to self-configuring circuitry. This is not a work on tool-development *per se*, and it is not the purpose of this work to expand on present knowledge in tool development, place-and-route techniques, and so on.

1.5 Organization of paper

The remainder of this paper is organized as follows:

- Chapter 2 presents an overview of a reconfigurable system that supports internal monitoring and configuration of its reconfigurable elements. While the techniques presented in this paper will be applicable to general systems that support self-configuration, working with a specific reconfigurable architecture will simplify the discussions.

- Chapter 3 discusses one particular application of internal control, for managing the initial configuration of a very large collection of homogeneous sub-circuits. This will also illustrate the usefulness of Hypothesis 4, as the synthesized configuration circuit will be used to synthesize the configuration circuit itself.

- Chapter 4 describes some general techniques for manipulating circuits from inside a reconfigurable device itself. The circuits and *sequences* described will be used throughout much of the remaining work described herein.

- Chapter 5 illustrates a specific application of these techniques: a massively-parallel system for simulating heat flow through a 2-D or 3-D medium, based on the 2-D and 3-D heat equation. This application will map a material being analyzed onto a set of reconfigurable elements. Efficient initial configuration of the system using the concepts presented in Chapter 3 will also be discussed.

- Chapter 6 will collect the results of the previous chapters, and compare them to similarly-sized systems implemented with other potential technologies.

- The paper concludes with Chapter 7, which summarizes the findings and discusses future work.

Chapter 2

Background

This chapter discusses architectural considerations related to the four main hypothesis of this paper. To re-iterate, these are:

1. Industry trends point to a continuing shift towards extremely large reconfigurable devices;

2. As the number of transistors in reconfigurable devices continues to increase, it will become increasingly useful if those transistors can be monitored and manipulated from inside the device itself;

3. Having that internal management circuitry itself be reconfigurable gives maximum flexibility; and

4. If the reconfiguration of management circuitry is indistinguishable from the reconfiguration of target circuits, then you eliminate hierarchy, and in some sense "close the loop."

These are fundamentally different from the standard CPU/memory model which utilizes a single (or, in modern cases, perhaps several) processors; is not scalable; and is comprised of fixed hardware. The following sections will discuss each of the above criteria, how they can be achieved, and what the consequences of them are.

2.1 Large-Scale Reconfigurable Systems

Traditional computing has long been based on the von Neumann model [**?**], comprised primarily of two components:

- a memory that stores both data and instructions; and
- a central processing unit (CPU) that performs calculations on the stored data, in accordance with the stored instructions.

While the basic design of a memory may be the same regardless of its capacity, it is generally not possible to simply "add more transistors" to an existing design to create a larger memory. While one can increase the width of the memory (i.e., the number of bits stored at each address), this is generally not as useful as extending the *length* of the memory, i.e., the total number of memory locations that can be addressed. Doing so requires not only a wider

address bus, but a more-complex address-decoding system for selecting the desired memory cells based on the address. The extension of a memory architecture to a larger capacity can be done in a fairly automated way, but it's certainly not as straightforward as simply connecting, say, 1,000 memories together side-by-side to create a larger memory.

Scaling of a CPU is also challenging. While it is possible to design small CPUs specifically to be scalable — as was done, for example, by AMD in their 2900 Family of bit-slice devices [**?**] — in general a CPU is not a scalable design. Given a pair of four bit processors (such as the Intel 4004), it's not clear how to connect them somehow to produce an 8-bit processor. Equally difficult is imagining how to simply double the number of transistors inside the device to achieve an 8-bit processor. Even by widening the registers, buses, ALU, and so on, one still would need to consider expanded addressing modes, more-sophisticated instruction sets, and so on (a 64-bit processor possessing the 4004's instruction set and addressing modes would probably be of limited usefulness).

This scaling difficulty is somehow inherent in the von Neumann model. This may be because it's based on a single-process; and while, given more transistors, it may be feasible to manufacture "more CPUs," there's no clear way to manufacture "more CPU."

Reconfigurable hardware [**?**] offers an alternative to the von Neumann CPU/memory model. Each is a general-purpose system, but with a basic difference:

- a CPU/memory system has fixed circuitry; its generality comes from its *programmability*, i.e., its ability to execute different algorithms on that fixed circuitry; whereas

- reconfigurable hardware's generality comes from the ability to change the circuitry itself.

At some level, reconfigurable hardware is more-easily scaled up, according to ones ability to design and utilize larger circuits (circuits containing more transistors, gates, etc.). However, there is still an architecture associated with reconfigurable devices: an organization of the "blank hardware" in a way that allows a configuration to be loaded into the system so as to achieve the desired functional behavior.

Of course, for presently-available reconfigurable devices, the lowest-level underlying hardware does not physically change when the device is configured. There is still a static base-layer of hardware, that, in some sense, interprets the desired configuration of the system. For example, a block might be specified to act as an AND or a NOR gate based on the value loaded into a nearby register.

Most modern reconfigurable devices, such as those by Xilinx, are configured by loading a *configuration string* into the device. This string has a specific (usually per-device-type) organization, and generally specifies the functional behavior of each reconfigurable element in the system, as well as their interconnection with each other.

While it may be straightforward to design a larger circuit (i.e., one that uses more hardware), the design of a larger reconfigurable device is not so straightforward, and designing a device that contains more reconfigurable elements may require re-architecting the entire system. Thus, while hardware designs may be "more scalable," the devices on which these designs

are implemented still do not scale very easily.

If we wish to utilize trillions of transistors (or 10^{23} transistors, as in an Avogadro machine), we need a base architecture that not only supports reconfiguration, but one that also scales up naturally, i.e., one which can utilize an ever-increasing number of available transistors without needing to be fundamentally re-designed.

2.1.1 A Scalable Design

For our purposes, a *perfectly scalable architecture* is one with the property that if two identical systems S are (in some way) connected to each other, the result is a larger version S' of that same system. The way in which this connection occurs may vary; what is important in this definition is that no changes need be made *inside* S in order to two (or more) of them together. Another way to view this is as follows: if one looks at a subset of S', it will be impossible to tell if one is looking at a section of S', or at the original system S. In some sense, a perfectly scalable architecture thus possesses a fundamentally *fractal* nature [**?**].

Devices such as those in the Xilinx Virtex series incorporate a sophisticated, hierarchical routing system for interconnecting configurable elements [**?**]. While this potentially simplifies generation of a configuration file and improves performance of the implemented circuitry, it is unfortunately an impediment to scaling the system: generally, the more components contained inside the device, the more-complex the routing system will be (there are other impediments to scalability as well, such as the internal configuration mechanism, that will

be discussed below).

Figure 2.1 illustrates an alternative routing system, based on the notion of *nearest neighbors*. In such a system, each reconfigurable element is connected to a fixed set of neighboring elements, according to an established system-wide topology. Such a scheme satisfies our definition of a perfectly scalable architecture. Of course, there may be factors such as propagation delay, power requirements, and so on that affect how scalable the system is: but from an *architectural* point of view, no changes are needed in the design in order to create a larger system.

Figure 2.1 shows a set of four elements connected in this planar, nearest-neighbor way. Figure 2.2 shows four such sets, connected along their edges, to form a larger (4×4) set of elements.

The nature of the reconfigurable elements must also be decided. There are many possibilities for what to endow these blocks with: memory, arithmetic units, signal processing subsystems, text processors, or complete CPU/memory systems are all interesting options, and depending on the nature of the problem being solved, each of these may be a good choice. However, since the goal is to architect a single, universal system, a simpler element from which more-complex units may be built will prove most-generally useful.

The simplest form of combinatorial processing is a simple map from inputs to outputs, e.g., a truth table. Such a map can consume a large amount of memory: for example, a 10-input 10-output device would require 2^{10} rows of memory, each 10 bits wide, for a total of 10,240 bits, assuming one wished to be able to represent **any** possible mapping from inputs to outputs.

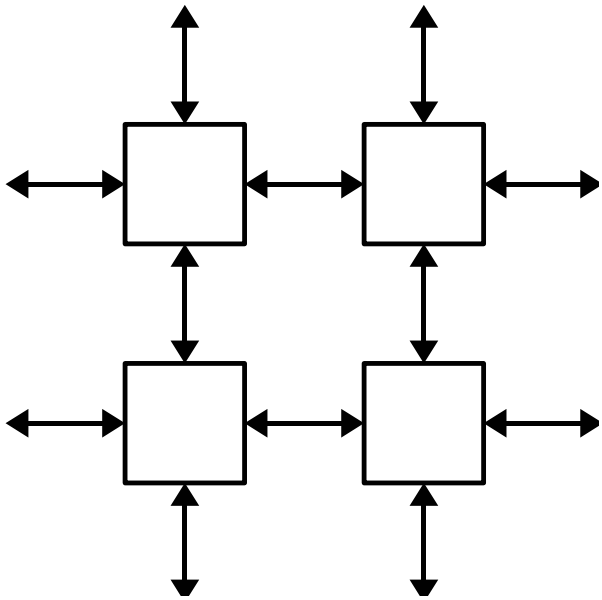

Figure 2.1: A 2×2 set of elements, networked in a nearest-neighbor topology. Each element communicates with a maximum of four neighboring elements (two in this example). Unconnected I/O lines are available to external devices.

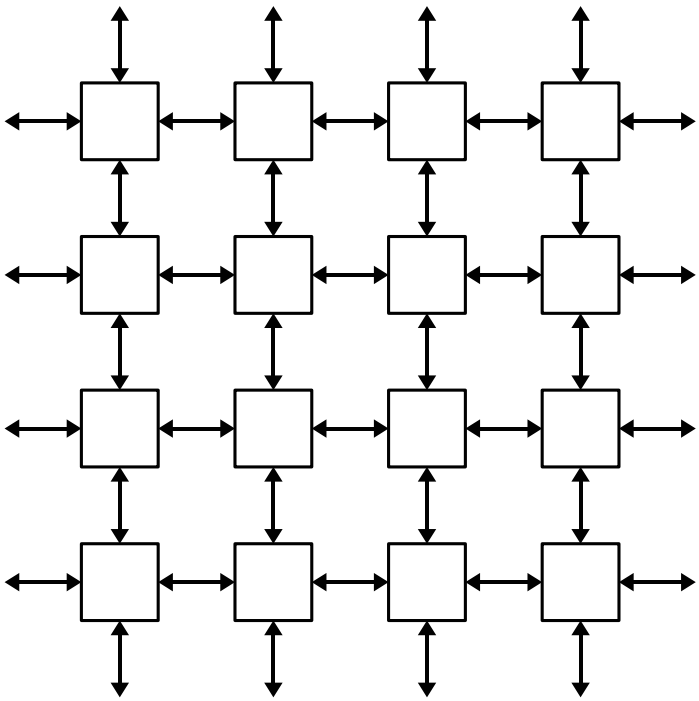

Figure 2.2: A 4×4 set of elements. This collection could be made by connecting four copies of the circuit shown in Figure 2.1 along their edges. Each 2×2 sub-collection is, in fact, identical to the circuit shown in Figure 2.1.

In general, this will probably be overkill. However, supposing a manufacturing capability where a trillion trillion transistors may be incorporated into a single device, the difference between 10 bits and 10,000 bits of memory per element becomes less critical. And, in fact, we don't need to work with 10-input/10-output devices. The simplest two-dimensional tiling of elements would be based on triangles: each element having three sides, and therefore being connected to three neighbors. If we let each element exchange a single bit with each of those neighbors, the element's truth table would thus be 2^3 rows $\times$ 3 columns, for a total of 24 bits. A perhaps more-natural tiling may be achieved with four-sided elements, raising the truth table size to 64 bits. If these elements exchange two bits with each other, the truth table grows to 2^8 rows $\times$ 8 columns=2,048 bits: a significant increase in size.

The simplest three-dimensional tiling of elements would be based on tetrahedral elements: each element thus having four sides, and therefore being connected to four neighbors. The complexity is identical to the two-dimensional, four-sided case, with each element requiring 64 bits. A more-canonical element would be a 6-sided cube, requiring a truth table of 2^6 rows $\times$ 6 columns=384 bits per truth table. Extending this system to passing two bits between each pair of elements, the truth table size grows to $2^{12} \times 12 = 49{,}152$ bits per truth table. Note that it's also possible to architect hyper-systems with dimensionality greater than three. Such systems, however, generally fail in their scalability, and thus are not considered herein.

Here again, while it may be more convenient to implement elements that exchange multiple bits with each other, or that connect directly to more neighboring elements, doing so is not actually necessary: one can view a collection of two or more elements as acting like a single

composite element. Given the rapid growth of truth table memory requirements, the present work will focus on a relatively simple compromise: four-sided elements, each exchanging a single bit with each of four neighbors. Some consideration will also be given later to six-sided elements, in discussions on three-dimensional circuitry.

Similarly, it will suffice for all elements to be identical to each other, with more-complex elements being built from collections of these simple elements. Note that under this scheme, elements may be used for any of several purposes:

1. to implement simple logic functions;
2. to implement more-complex logic functions, by working in tandem with other nearby elements;
3. to implement memory units such as flip-flops, again by working in tandem with other elements; and
4. to act as a *wire*, i.e., to pass information between non-adjacent elements.

There are additional uses for elements, including using their truth tables as a higher-density memory. This will be described later.

Figure 2.3 shows the design of a simple truth table-based four-sided reconfigurable element. The truth table is stored directly inside the element's internal memory, and is used to set output values based on inputs going into the element. Note that the element operates asynchronously: there is no clocking inherent in its operation. As soon as an input changes

value, the outputs will change accordingly (after, of course, a slight propagation delay as the circuitry computes the new output values). Missing from this figure is any circuitry for actually loading the truth table into the element's memory. This will be covered in the next sections below.

Figure 2.4 shows a small collection of these elements. Note the following:

- this collections conforms to the scalability requirements described above;
- a single element can implement any combinatorial four-input four-output function, including of course basic logic functions such as AND, OR and NOT;
- elements can be used to implement wires; and
- a single element may be used to implement a pair of *crossing* wires, allowing data to be passed in two orthogonal directions without interference.

Thus, given a sufficiently-large collection of elements, it should be possible to configure them to implement any circuit that could be build from interconnected gates on a two-layer substrate. The system is thus useful for building extremely large circuits, such as those that might be used for large search problems, or for simulation of large two-dimensional systems. Chapter 3 will explore such an application in detail. More-generally, large two-dimensional circuitry may be useful for implementing massively-parallel systems, while avoiding the limitations found in overly-constrained pre-wired collections of CPU/memory nodes.

The following sections will further develop this basic element's design, by adding the neces-

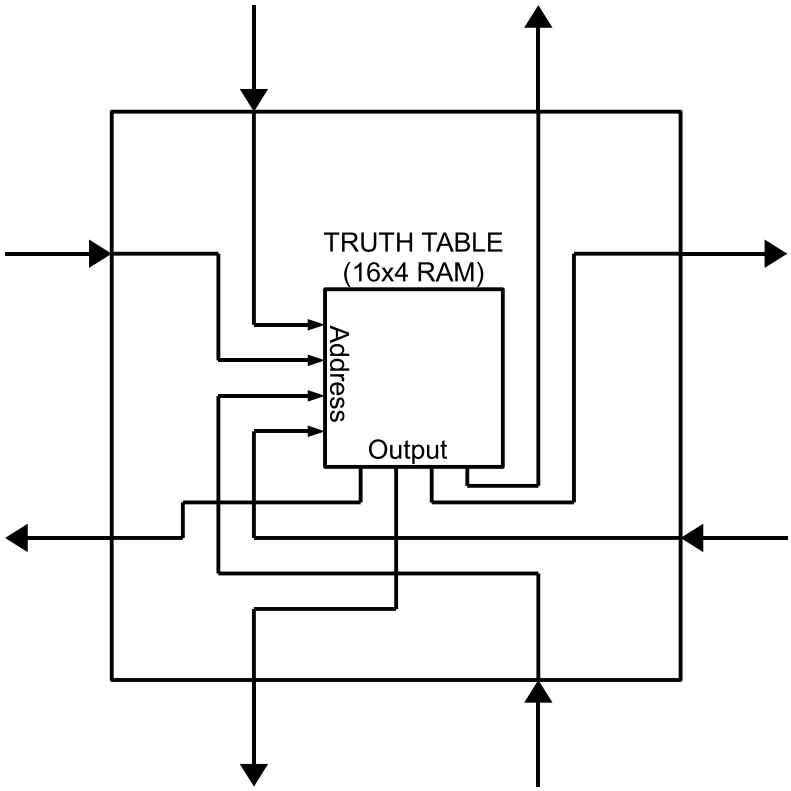

Figure 2.3: A single four-sided reconfigurable element based on a 4-input 4-output truth table. The truth table is simply a 16x4 RAM that stores, for each possible combination of the element's four inputs, the precise set of four outputs to be generated. The element can thus be made to perform any desired static mapping. Functions that can be implemented include: logic gates; 2-1 multiplexers; 1-bit adders; and simply wires for routing data among other elements.

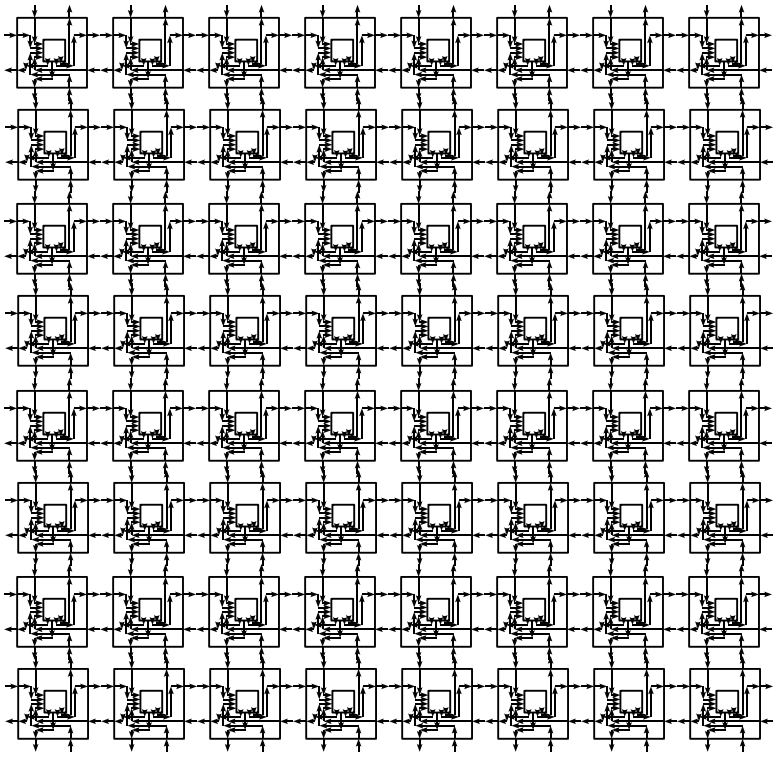

Figure 2.4: Collection of reconfigurable elements such as the one shown in Figure 2.3. By configuring each element to be a particular logic element, piece of a wire, or small-scale function, complete digital circuits can be implemented on this collection.

sary circuitry to allow an element's truth table to be loaded with a desired set of bits.

2.2 Internal Control

Traditional reconfigurable devices are generally configured from *outside* the device itself. Typically this involves loading a bit-stream into the device. The bit-stream is routed throughout the various memories and registers inside the device, and controls the configuration and interconnection of the reconfigurable elements. While this works well for a device with a small number of components, it is probably not sufficiently powerful for configuring an extremely large device. Logically, a device containing 10^{23} reconfigurable elements, each element requiring n specification bits, would require a configuration string of at least $n \times 10^{23}$ bits. Most likely, there would be difficulties in managing a configuration file whose size is on the order of *a trillion terabytes*. Moreover, even assuming a very fast processing time of, say, one pico second per n bits, processing $n \times 10^{23}$ bits would require 10^{11} seconds — more than 3,000 years. In general, externally managing and controlling systems even at the tera-scale can be quite challenging [**?**].

An alternative approach would be to control it from *inside* itself, i.e. to configure the elements from a set of distributed, local configuration controllers within the reconfigurable device. To support this approach, we can modify the design of a basic reconfigurable element as shown in Figure 2.5. Here, a pair of additional inputs — C and B — have been added to the element. The C input is a flag, indicating that the element is being configured. When

C is asserted, the element's new truth table is loaded through the B input.

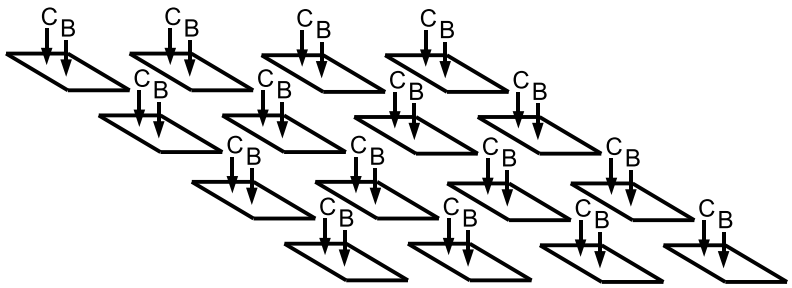

Figure 2.5: Reconfigurable elements augmented with configuration inputs. The C input is asserted to initiate a configuration operation on the given element. The B input is used to send in the new truth table bits, as a serial bit-stream. A system-wide external clock is assumed. This figure illustrates a 4×4 set of such elements. For clarity, connections between each element and its four immediate neighbors are not shown.

Rather than making B very wide, it can be a single bit, and the truth table is loaded serially. This requires some kind of clock for sampling the bits on B. A single system-wide clock can be used: it can be ignored when $C = 0$, and used for loading truth table bits when $C = 1$.

Note that the use of a single system-wide clock is not the most–efficient approach in terms of power consumption, generation of electrical noise, and so on. There are also technical hurdles associated with such a clock, including massive fan-out requirements, as well as its potential for being a critical failure point: if the clock were to be shorted to ground

anywhere in the system, the entire clockline may become unusable. There are ways to address some of these concerns, including employing locally-generated clocks that drive a sub-array of elements; selective disabling of clock-generation; or using a non-wired system for clock delivery (i.e., optical clocking). Such embellishments, while potentially critical to the practical development of an extremely–large–scale system, do not effect the basic use of such a system as discussed herein, and thus are not considered in this present work.

There are many ways to implement such a configuration circuit: one of the simplest would be to make the truth table memory a shift register; tie the SHIFT input to the system-wide clock; connect C to the SHIFT ENABLE input; and connect B to the D_{in} input, as shown in Figure 2.6.

There are several advantages to having this kind of configuration system:

- elements can be configured in parallel, thus potentially greatly improving configuration time for the entire device;
- partial reconfiguration is supported — in fact, it is the "default" configuration mode, since configuration of one element is independent of other elements;
- run-time reconfiguration is certainly supported, though in fact there is no such thing as a system-wide "run time" vs "configuration time" — rather, these are only per-element concepts;

Some sort of circuitry will still be required in order to utilize this configuration system to

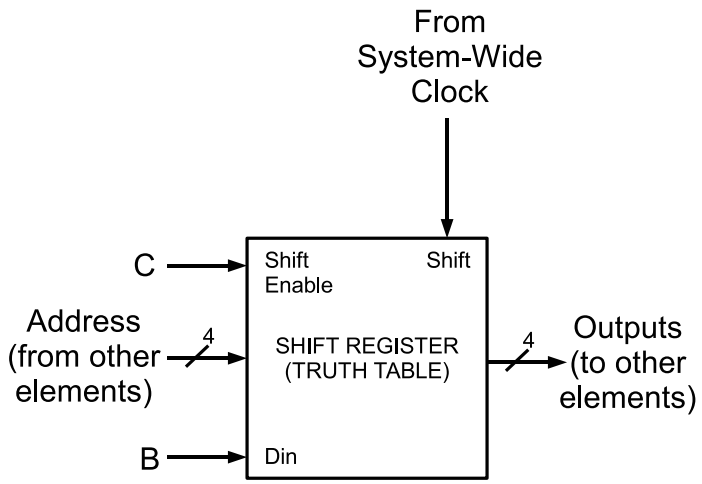

Figure 2.6: One possible implementation of a configurable element. The shift register stores the cell's truth table, which is loaded by asserting the C input, and then sending the desired bit pattern into the B input (synchronized with the system-wide clock). The register also needs to be able to act as a 16x4 RAM.

bootstrap the device. Figure 2.7 shows one example of such circuitry. In this case, the device will be bootstrapped with a repeating pattern of sub-circuits, each comprised of a 10×10 set of reconfigurable elements. Still assuming n bits of configuration information per element, one 10×10 sub-circuit requires $100n$ bits to specify its configuration. This (relatively small) bit-stream is loaded into the device from outside, and is distributed internally to each 10×10 sub-circuit, with each sub-circuit being configured in parallel with every other sub-circuit (not shown in Figure 2.7 is the circuitry to distribute the bit-stream's bits to each of the 100 elements). Chapter 4 will cover this approach in more detail, and will use it to efficiently bootstrap a circuit composed of a number of identical regions.

This particular example of a bootstrap circuit shows just one possible use of an internal, local, distributed configuration system. There are many possible applications, particularly if the configuration circuit is allowed to *read* an element's truth table as well as write it. Possible uses include:

- efficient bootstrapping of circuits containing a great deal of homogeneity;
- re-wiring of circuits in response to faults or changes in circuit requirements (size, etc.); and
- support for run-time compilation of hardware, such as just-in-time compilation of algorithms into hardware.

This capability for internal, local, distributed configuration also has potential for supporting management of hardware resources in the same way an operating system manages memory,

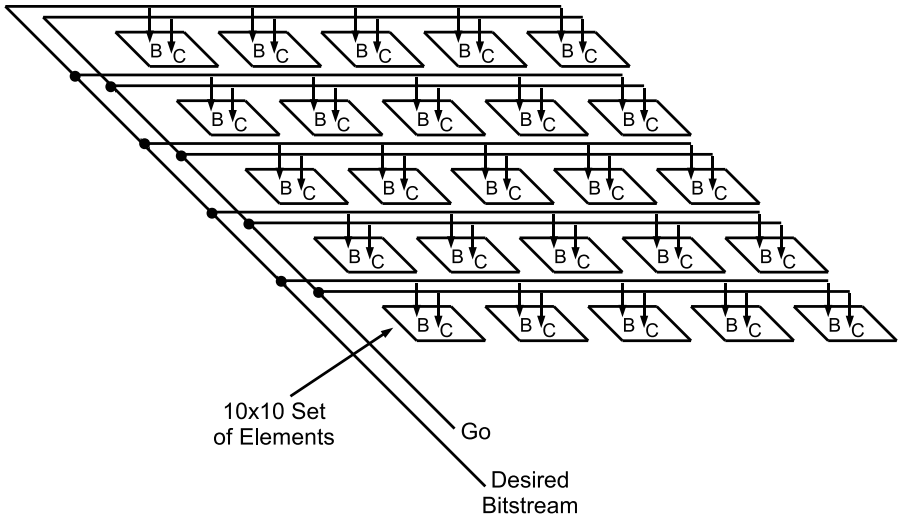

Figure 2.7: Sample Bootstrap Circuit. Each square is a 10×10 set of reconfigurable elements. The goal is to configure each 10×10 set identically. The B and C inputs are used to load a serial bit-stream that contains 100 truth tables (one per each element in the set). Since B and C are shared by all 25 sets, a single transmission of the Go and Desired Bit-stream signals will configure all sets simultaneously. This is one example of a sometimes useful configuration circuit. Omitted from the figure is the circuitry inside each 10×10 set of elements that is responsible for distributing the bit-stream to each of its 100 elements.

including:

- element re-use, by monitoring how long it's been since an element's inputs have changed;
- swapping of circuits in and out of physical elements, in support of a multi-processing model where several circuits could be sharing one physical device, each being swapped in to execute for a small slice of time before being swapped out; and
- implementation of *virtual hardware*, to potentially allow implementation of circuits that do not physically fit on the hardware device.

None of this, however, is feasible with the simple control system shown in Figure 2.7. The problem is that each configuration system described above *requires different circuitry in the control layer*. And while it's possible to fabricate devices with different circuitry, a more-generally-useful approach is to *construct the control layer from reconfigurable elements*.

2.3 Configurable Control Systems

Figure 2.8 shows a slightly-modified reconfigurable device, similar to Figure 2.5, but with one difference: the control layer itself is now also reconfigurable. In this case, it is composed of a regular collection of reconfigurable elements, each connected to a fixed set of nearest neighbors. This control layer is very similar to the original reconfigurable substrate (called the "base layer") described in the first section of this chapter. Each element can be configured to perform simple input-to-output mappings, based on a truth table stored inside itself.

By properly configuring these elements, it's possible to implement a wide range of digital circuitry on this control layer, including circuits that perform somewhat non-traditional functions such as those described above (e.g., automatic re-wiring of circuitry based on run-time events).

In Figure 2.8, each element in the (reconfigurable) control layer is responsible for configuring one element in the base layer. This means it must be able to control the base layer element's C and B inputs. This can be accomplished by adding two additional outputs to the control layer elements' truth tables. Thus, for each combination of four inputs (one from each neighbor), 6 outputs are produced: 4 being sent to the element's neighbors; and two being used to control the base layer's element.

This relatively simple scheme allows circuits built on the control layer to manipulate circuits on the base layer with a high degree of generality. Even greater control can be established by allowing the control layer to *read* information from the base layer: one or more of each base layer element's output bits can thus be provided as further inputs to the control layer element's truth tables. Bits being overwritten in each element's truth table can also be provided to the control layer, thus allowing the control layer to *read* configuration information from the base layer.

With a control layer such as this, one can envision, for example, circuits that detect faults in the base layer and respond by re-wiring affected circuitry; or, more-generally, control circuits that monitor base-layer circuitry, and respond to particular conditions in precisely-controlled ways. This opens the pathway to true dynamic circuitry, where, in some sense, *hardware can*

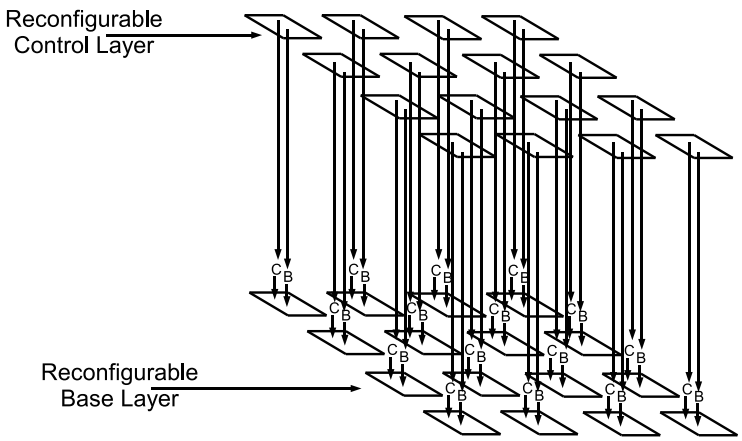

Figure 2.8: Reconfigurable Control Layer for Managing the Reconfigurable Base Layer. The control layer is composed of a series of identical reconfigurable elements, each connected to its immediate neighbors in the control layer plane (for simplicity, the connections among elements within each plane are not shown). The control layer is able to assert a pair of outputs — B and C — that are connected to the B and C inputs of a corresponding base layer element. The control layer can be used to build arbitrary digital circuits that manage the configuration of the base layer.

be processed as simply another data type. Chapter 5 will explore the details of such circuits for dynamically manipulating circuitry entirely from inside the system itself.

There are two lingering issues with the scheme presented above:

1. the elements of the control layer are already fairly complex, requiring perhaps 9 inputs and 10 output per element; and, more-critically
2. there remains the question of *how to configure the control layer itself.*

It is this second issue that is most-important: how will the control later be controlled? One approach is to add a *super-control later* that is responsible for configuring each element in the control layer. But this raises the same question: what will that super-control layer look like? Will it be composed of fixed circuitry, or will it itself also be reconfigurable? And if the latter, what will control *its* configuration. This is the topic of the next section (and it will also address the first issue above).

2.4 Hierarchy-Free Organization

A brief seeming diversion may make the main points of this section more understandable.

2.4.1 Subject/Object Non-dualism

If a tree falls in the woods, and no one is there to hear it, does it still make a sound [**?**]? This well-known *kōan* raises interesting questions. Of course, a tree falling in the woods will still disturb air molecules, regardless of the presence of an observer. The nature of "sound" may confuse the discussion though. A different, more-easily analyzed version of this kōan might be: "If there's no one available to eat a hot pepper, is it still spicy?" At first, this may sound like a foolish question: peppers don't suddenly become spicy simply because someone approaches them. Or do they?

Hot peppers taste spicy because of the presence of a chemical:

$$(CH_3)_2CHCH = CH(CH_2)_4CONHCH_2C_6H_3 - 4 - (OH) - 3 - (OCH_3)),$$

more-easily called *capsaicin* [**?**]. It is the interaction of this chemical with certain sensory neurons in humans that leads to the spicy, or burning sensation experienced when tasting a hot pepper [**?**]. Certainly, the concentration of capsaicin is independent of the presence of someone to taste the pepper. However, *the interaction of this chemical with the taster's receptors* clearly **does** depends not only on the presence of a taster, but on the *nature* of that taster.

Put simply, what tastes spicy to one person may taste bland to another. More-profoundly, some animals (e.g., birds) do not have receptors that bind with capsaicin, and thus, as far as is understood, do not perceive hot peppers to be spicy or "hot." In view of this, perhaps this modified kōan seems a little less foolish.

In fact, this is an example of a more-general phenomenon. It's common to think of a characteristic such as "spiciness" as being an innate property of an object, as *belonging to* the pepper; and a taster is one who simply observes that characteristic. This is an example of *dualism*, in that it views the observer as distinct from that which is observed [**?**]. As this example suggests, a dualistic view of the universe may hide deeper, more-subtle facts.

The term "taste" in the original question has embedded in it an assumed observer, and an assumed mechanism by which the observer will analyze the food being tasted. In more-general terms, "taste" might simply refer to certain types of chemical interactions that occur in, say, the human tongue in the presence of certain chemicals. But under such a definition, it may not be unreasonable to speak of a pepper as "tasting a human tongue" when it comes in contact with it — after all, there are chemical changes that take place inside the pepper when it contacts a tongue (though, as far as we know, there is no processing of that phenomenon inside a "pepper mind"). In a pure non-dualistic view of the world though, speaking of interchanging the observer with that which is observed might be a misnomer: a more-accurate analysis might be that *there is no distinction between the observer and that which is observed.*

Dualistic views are problematic in more than philosophical ways. Consider the "liar paradox" [**?**]. In one form, a speaker makes two statements:

1. "Everything I say is a lie" and then

2. "I am now lying."

The question is then posed: is statement 2 true or false? There are seemingly only two possibilities:

1. statement 2 is true. But this would contradict statement 1, which says the speaker never states the truth. Or
2. statement 2 is false. But that means the speaker was **not** lying, and thus told the truth, which contradicts statement 1.

The liar paradox can be directly applied to computer science to create an unsolvable problem: the Halting Problem (HP) [**?**]. The goal of HP is to write a computer program P that can analyze another program Q and tell you whether or not Q ever halts (executes a HALT statement). Presumably, such a program P would contain code that prints the statement "Q does halt" as well as code that prints the statement "Q does NOT halt." Given such a program P, one modifies it to create a new program P', with the following changes:

- whenever P' prints "Q does halt," it then enters an infinite loop; and
- whenever P' prints "Q does NOT halt," it then executes a HALT statement.

What is going wrong here? How can such a seemingly simple problem be impossible? If one were to go about trying to code a solution to HP, one might begin by writing an interpreter for the programming language in question: building a virtual machine that executes P, while analyzing its behavior, looking for repeated loops with identical states, or of course for execution of a HALT statement. By the above analysis, the difficulty begins when P' is

ingested by the interpreter. Remember, in this instance, P' is a slightly modified version of the interpreter. So the interpreter will be trying to interpret itself. That's not a problem *per se*, but P' requires input to operate, and the input is P'. In other words, the interpreter needs to interpret a copy of itself interpreting a copy of itself. This leads to a kind of infinite descent, under which one may more easily believe that certain seemingly simple tasks are in fact impossible. And, in some sense, this comes back to dualism: there is an observer (P') being asked to analyze itself, but in fact it is not analyzing itself, *it is analyzing a* ***copy*** *of itself.* And somehow, this seems to be a critical distinction.

Here is a more-concrete example. Consider the question of writing a computer program that prints out its own source code (and we want this program to be self-contained, i.e., we don't simply want to open the source file on disk and copy the code from there). Let's assume a simple BASIC-like language. We might begin with something like:

```
10 PRINT "10 PRINT
```

This will print the first 8 characters of the program. Next we need to print the quotation mark. Let's assume two consecutive quotes represent a single character. Our program now looks like:

```
10 PRINT "10 PRINT ""10 PRINT
```

Now our program will print the first 18 characters of itself; but next we need to print “”””. We can continue writing the program as:

```
10 PRINT "10 PRINT """10 PRINT
```

but it's clear what's happening: we're solving part of the problem, but as we solve it, we also cause the problem to grow larger. And in this way, we will never actually finish writing the desired program.

A non-dualistic approach will solve the problem though. Here's a C-language program that prints out an exact copy of its source code:

```
static char p[20][1024]={
"#include <stdio.h>",
"#include <string.h>",
"",
"main()",
"{",
"       int i,j;",
"",
"       printf(\"static char p[20][1024]={\\n\");",
"       for (i=0;i<20;i++){",
"               if (i>0) printf(\"\\\",\\n\");",
"               printf(\"\\\"\");",
"               for (j=0;j<strlen(p[i]);j++){",
"               if ((p[i][j]=='\\\"' || p[i][j]=='\\\\')) printf(\"\\\\\");",
```

```
"               printf(\"%c\",p[i][j]);",
"         }",
"       }",
"       printf(\"\\\"};\\n\\n\");",
"",
"    for (i=0;i<20;i++) printf(\"%s\\n\",p[i]);",
"}"};


#include <stdio.h>
#include <string.h>


main()
{
       int i,j;


       printf("static char p[20][1024]={\n");
       for (i=0;i<20;i++){
               if (i>0) printf("\",\n");
               printf("\"");
               for (j=0;j<strlen(p[i]);j++){
               if ((p[i][j]=='\"' || p[i][j]=='\\')) printf("\\");
```

```
            printf("%c",p[i][j]);
          }
        }
        printf("\"};\n\n");

    for (i=0;i<20;i++) printf("%s\n",p[i]);
}
```

There are two aspects of this program that make it work. The first is that a copy of (part of) itself is embedded inside itself. The program has two main sections: the first section is a copy of code stored as *data*; and the second part is almost-identical code that will be treated by a compiler as *code*. The trick here is that the data copy is sufficient for reproducing both the data and the code copies. The program actually acts similar to the replication process for DNA: it reads a set of instructions, and produces two copies, the first being an identical copy (transcription) and the second being interpreted as to what those instructions mean (translation) [**?**]. The second aspect of this program is the use of a backslash ("\") to escape certain characters. This is a local version of controlling the interpretation of something as code or data. For example, a double quotation mark may indicate to the compiler that a string is beginning or ending (that's interpreting it as a piece of code); or, preceded by \, it is simply another ASCII character (i.e., it's interpreted as data).

2.4.2 Application of Non-Dualism to Architectural Design

Bringing all of this back to the question of how to organize the base layer and the control layer, some similarities may be observed. In trying to differentiate between circuitry that *controls* and circuitry that *is controlled*, we've encroached on the issue of hierarchy (a form of dualism). And in imposing this hierarchy on the system, a problem similar to the infinite descent of the Halting Problem solver has appeared. Namely, the base layer can be controlled, but to control the control layer requires addition of another layer (the super-control layer), and controlling *that* layer requires a still further layer, and so on.

What would a non-dualistic organization look like? One in which, in some form, the base- and control-layers were not simply intertwined, but were in fact *identically the same thing*. The way this can be achieved is to endow each element in the base layer with the ability to read and write the configuration of elements in the base layer. And the way to achieve this is to use something similar to the \ from the example of the program that prints out its own source code.

Figure 2.9 shows a modified (base-layer) reconfigurable element. As before, it has four outputs and inputs (labeled "D"), available to and from four immediate neighbors. This element also is capable of producing C outputs to each of those neighbors, i.e., this element can configure any of its neighboring elements. Moreover, rather than have separate B lines for reading and writing that configuration, the same D lines can be used, but simply *interpreted* differently, based on the value of an element's C input (so the C line is the analog of the \).

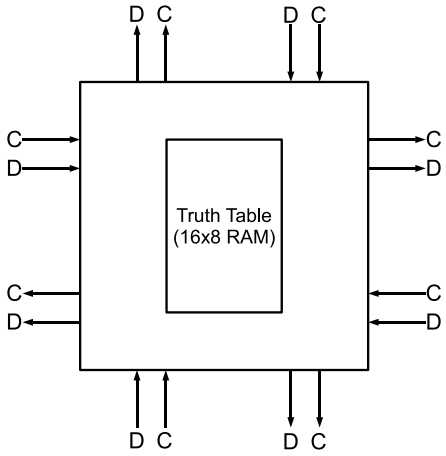

Figure 2.9: Modified Reconfigurable Element. This element is now able to control any neighboring element's C and B inputs in order to configure that neighboring element. The B input is actually shared with the D input: the meaning of the D input is determined by the value of the corresponding C input. The truth table still has 16 rows (one for each of the possible combinations of 4 D inputs), but its width is now 8 bits (4 D outputs and 4 C output).

The element's truth table thus has four D input (one from each neighbor); 4 C outputs (each responsible for signaling the reconfiguration of a neighbor); and 4 D outputs (each of which is interpreted as either data, to be loaded verbatim into the receiving element's truth table; or as code, to be "executed," i.e., used in mapping inputs to outputs. And the interpretation of a D input is based in the value of the corresponding C input.

Figure 2.10 shows an example of an element's truth table.

As before, one may assume a system-wide clock, used for shifting in D bits when an element is being reconfigured. Also, the D outputs from an element can be used to present the bits of the element's truth table as they are about to be overwritten. In this way, any element can both read and write the configuration truth table of any of its neighboring elements.

In summary, the operation of an element may be described as follows:

- if all C inputs are 0, the element is operating in a data-processing mode (called "D mode");
- if any C inputs are 1, the element is in a mode where it is being configured (called "C mode");
- in D-mode, C and D outputs are produced based on the pattern of D inputs and the contents of the element's truth table;
- in C-mode:
  - C outputs are set to 0 (this is simply a convention that has proven useful);

INPUTS				OUTPUTS							
DN	DS	DW	DE	CN	CS	CW	CE	DN	DS	DW	DE
0	0	0	0	b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	1	b15			...				b8
0	0	1	0				...				
0	0	1	1				...				
0	1	0	0				...				
0	1	0	1	.			...				.
0	1	1	0	.			...				.
0	1	1	1	.			...				.
1	0	0	0				...				
1	0	0	1				...				
1	0	1	0				...				
1	0	1	1				...				
1	1	0	0				...				
1	1	0	1				...				
1	1	1	0				...				
1	1	1	1	b127			...				b120

Figure 2.10: Sample 128 Bit Truth Table. Inputs and outputs have been labeled with compass directions ("N," "S," "W" and "E") to indicate which neighbor they are connected to. For each combination of D inputs, 8 outputs are specified. The collection of 128 outputs bits (b0–b127) constitute the complete identity of an element. Not all 128 outputs are explicitly shown in this figure.

  - on sides where $C_{in} = 0$ D outputs are also set to 0;
  - on sides where $C_{in} = 1$ D outputs are set to the truth table bit that will next be overwritten; and
  - on sides where $C_{in} = 1$, D inputs are used to read bit values that are written into the truth table (if there are multiple sides where $C_{in} = 1$, the D inputs are ORed together — another useful convention).

All truth table read and write operations are synchronized by a single system-wide clock. This clock can be used to generate synchronized clocks inside the system, by:

- loading a repeating bit pattern into one element's configuration memory;
- placing that element (called a "crystal") into C-mode;
- re-loading the crystal's configuration memory (i.e., performing a non-destructive read); and
- using the read pattern as the desired internally-generated clock pattern.

The essential aspect of this architecture is that all elements are equal in their ability to control or be controlled by other elements. There is no distinction between two types of elements: any element may assume either role, and that role may change over time. This feature will find powerful application in the remaining chapters. In essence, this introduces *scalability of the* ***architecture***. In other words, since the controllers are exactly the same elements as the elements they control, we can scale up the number of elements without

necessarily creating a system so large it can't be controlled. In effect, as we scale up the array of elements, we also scale up the array of controllers.

Figure 2.11 shows some simple circuits for reading an element's configuration, and copying it to another element. This is a form of *element replication*, and can be used as a basis for on-the-fly circuit synthesis.

Figure 2.12 is similar to the last circuit in Figure 2.11, but with one change: it copies the truth table into multiple target elements, thus making several copies of the original element. This is a building block for circuits that perform *parallel configuration*.

Figures 2.13 – 2.15 show a much more sophisticated circuit, based on the element replicators described above. This circuit is completely self-replicating. In fact, this circuit is the hardware equivalent of the program discussed earlier that prints out its own source code. Figure 2.13 is a sketch of the entire circuit, which consists of two main blocks: the MAIN GRID (the small square in the upper-left); and the EXPLODED GRID (the large rectangle on the right hand side of the figure). The circuitry on the right is merely a copy of the main grid, but with space included between each row of elements (hence the name), and with certain C inputs activated to prevent certain elements from performing configuration operations themselves. The main circuit on the left reads this copy and produces two outputs: one a "live" copy of the circuit (i.e., one that will actually "run"); and another that is a percent copy of the copy. Again, these correspond roughly to translation and transcription, respectively. In Figure 2.13 the main circuit has begun to build copies of itself and the exploded grid. The top of those copies can be seen at the bottom on the figure.

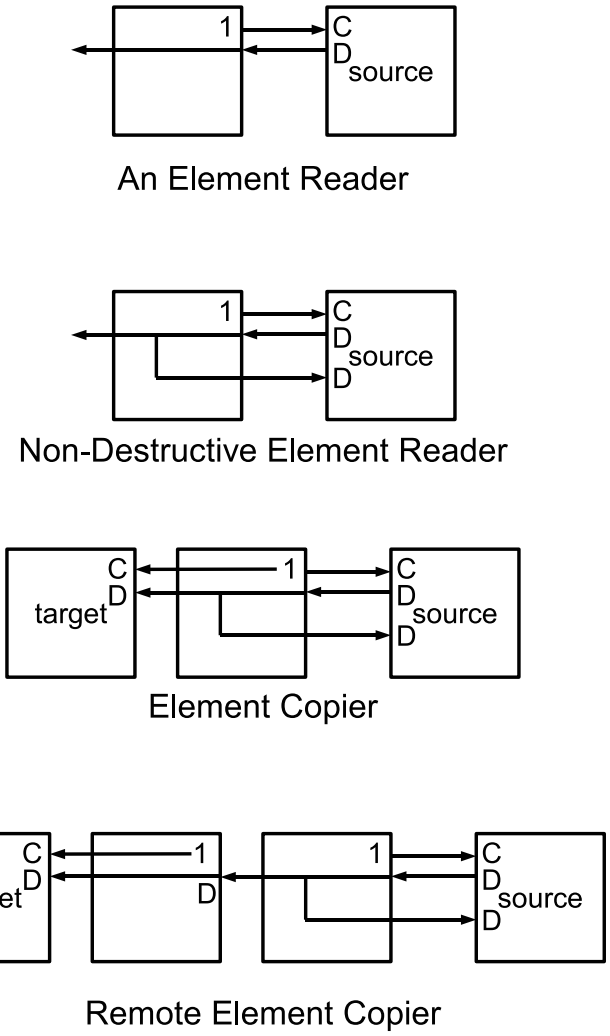

Figure 2.11: Circuits for Reading Elements. The basic two-element circuit at the top places a source element (on the right) into C mode, reads its truth table, and sends it out its western edge. The truth table is replaced with 0s. The non-destructive element reader improves on this design by restoring the truth table as it is read from the source element. The element copier sends the source's truth table bits into an adjacent element that is has also placed into C mode. The source's truth table is thus replicated inside the target element, thereby creating a copy of the source. The remote element copier extends this circuit further, but using an intervening element to route the truth table bits to a non-adjacent target element.

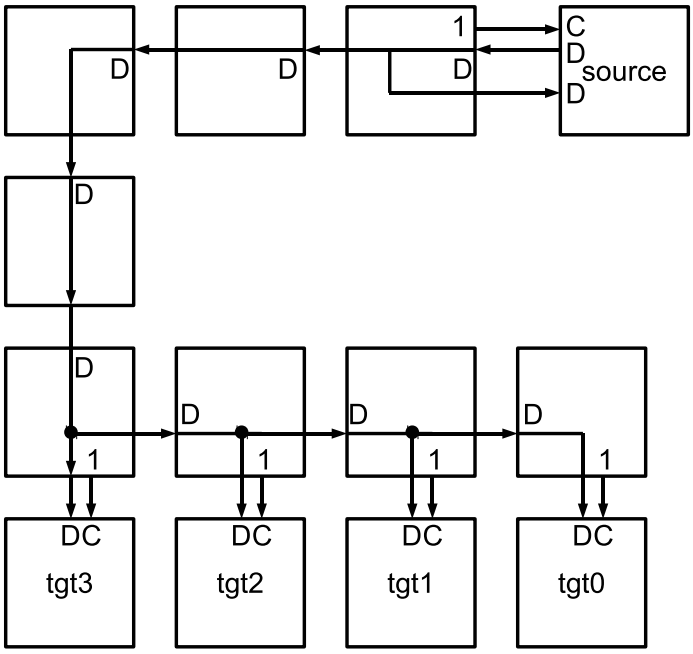

Figure 2.12: A parallel replication circuit. The source element's truth table is read, sent to the west, then to the south, and then distributed to the east to four elements, each located directly above a target element (labeled "tgt0"-"tgt3"). Each target element is placed in C mode by the element above it. Whatever truth table is located in the source element will be copied into the four target element. Moreover, this replication occurs *in parallel*, with all four target elements being configured at the same time.

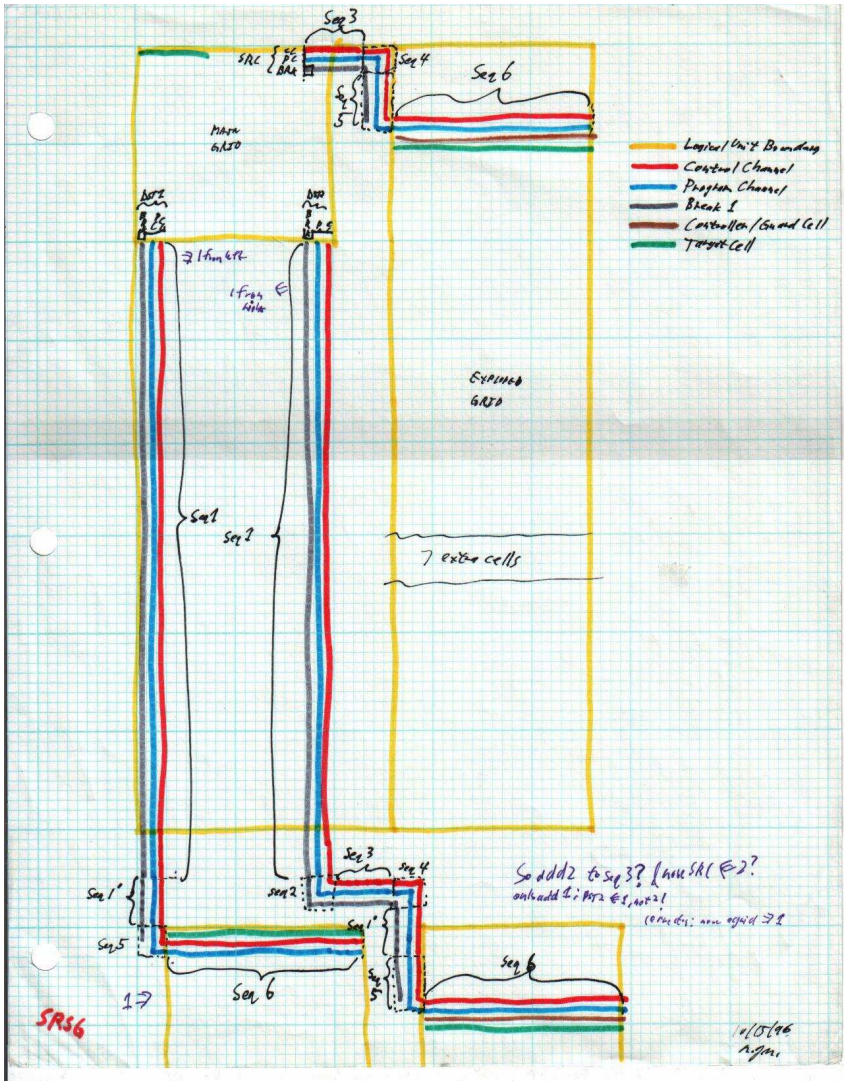

Figure 2.13: Sketch of a self-replicating circuit. The main grid (in the upper-left corner) reads a copy of itself stored in the exploded grid (right side of the figure). The exploded grid is very similar to the main grid, except that there is space between each row of elements, and certain elements are prevented from executing by the use of special control elements. The information that's read is used to create, to the south, two copies of the main grid: one that is an exact copy, and another that is an exploded copy. In this figure, the main grid has already begun making copies to the south. When it finishes, there will be an identical copy of the original circuit to the south.

Figure 2.14 shows an actual layout of the main circuit, using a graphical layout tool developed to design circuits on for this architecture. Figure 2.15 shows part of the circuit layout for the exploded grid. Here it is easy to see the rows of unused elements between each row of actual circuity. One can also observe that certain special control elements (one of which is circled in the figure) appear sporadically. These are used to prevent the element below them from actually performing its configured operation. They are, in effect, the hardware equivalent of a backslash ("\") in the self-replicating program described earlier: they cause the element below them to be treated as a piece of *data* rather than treated as a piece of hardware. [**?**] discusses in more detail an architecture such as has been proposed here.

The next chapter will discuss a particular scalable, large-scale problem, well-suited to the reconfigurable architecture presented here. The following chapter will discuss bootstrapping (how to initially configure) of that circuitry.

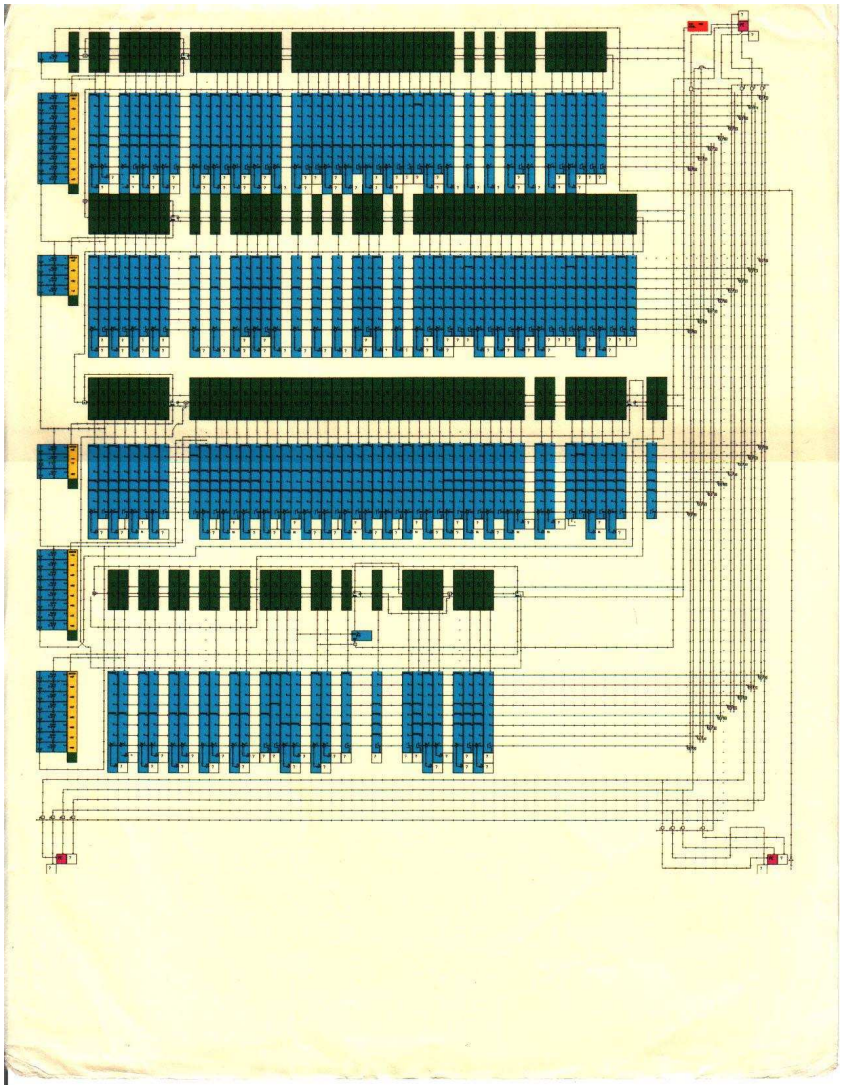

Figure 2.14: Detailed layout of the main grid of the self-replicating circuit shown in Figure 2.13. Each small square represents a single element. This main circuit is comprised of 85×75 elements.

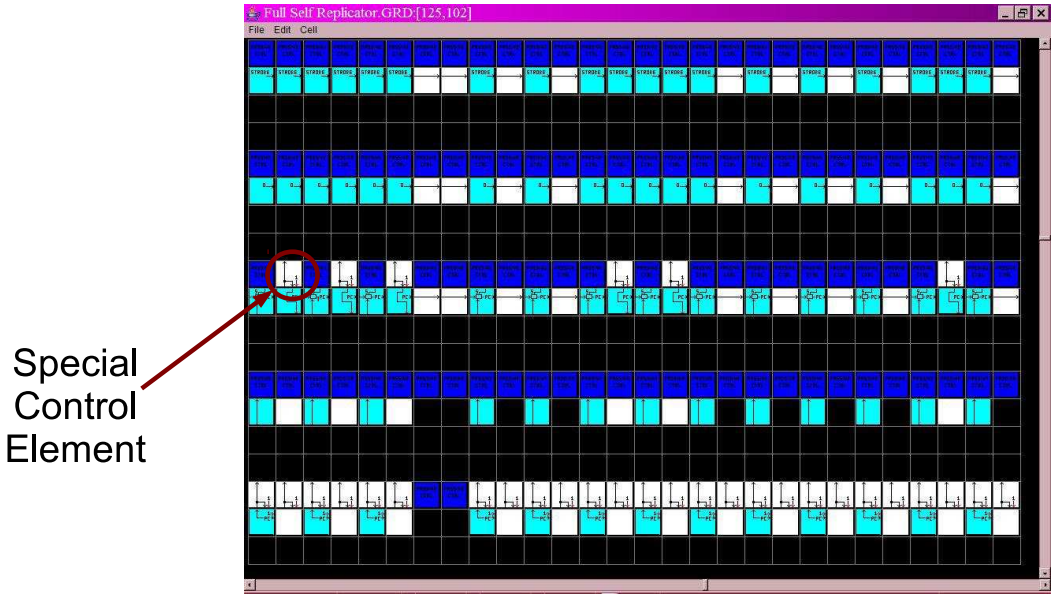

Figure 2.15: Blowup of part of the exploded grid. The first row consists of elements that read the source elements in the second row. These source elements correspond to the first row of the main grid. The third and fourth rows consist of initially-unused elements. These will be used to build wires so that the main grid can read the truth table from the source elements in the exploded grid. This four-row pattern repeats, and stores the second row of the main grid. Special readers can be seen in the 9^{th} row: these are special control elements that prevent their source element from potentially configuring nearby elements.

Chapter 3

Parallel Fault Testing and Configuration

Faced with the potential of, perhaps, trillions of configurable cells from which to build a target circuit, the question naturally arises: *how* will one design such a large system? Certainly, it may be possible to design a single circuits that utilize trillions of cells, but one can expect the cost of doing so — both in terms of time and of money — to be quite high. A more-likely scenario is to design systems that are comprised of a repeated pattern of smaller sub-circuits.

There are numerous examples of problems that can be solved efficiently using such a system: simulation of cellular automata; some kinds of image processing; and finite element analysis, to name a few. It is the latter of these — finite element analysis (FEA) — that was chosen as a driving application for this work. FEAs can be used to solve a number of important

problems that are very easy to state and simulate, but for which it is probably impossible to find a closed-form solution [**?**]. The simplicity of the Finite Element Analysis approach, combined with its natural spatial layout and nearest-neighbor connectivity among elements, make it an ideal target application for the work presented in this chapter.

3.1 Configuration Issues

In a general-purpose reconfigurable system, there needs to be some way to configure the programmable elements of the system, in order to implement our desired circuit. Typically, an FPGA includes programming pins that allow a *configuration bit-stream* to be sent into the chip. Internal circuitry decodes the bit-stream, and uses its contents to set the values of internal truth tables, multiplexers, and so on. Each FPGA has its own (usually) unique configuration mechanism, based on the size and layout of the chip's internal circuitry.

One potential problem as reconfigurable devices scale to the trillion cell size is how to efficiently configure them. Traditional FPGAs use a serial configuration bit-stream, which is processed by the FPGA's configuration mechanism one bit (or perhaps a small number of parallel bits) at a time. Logically, an FPGA with a trillion reconfigurable elements would need at least n trillion bits of configuration information, where n is the number of bits required to configure a single element. Even with a very fast pipeline, the time to simply send that much information into a system is likely to be considerable. If we envision an *Avogadro-scale device* (one containing 10^{24} elements), then configuration would be akin to

serially transferring the contents of *one trillion one terabyte drives*.

If we take a step back, we remember that a configuration bit-stream is nothing more than information. It's not bits *per se* that need to be transferred into the device; the device only needs to receive the *information* necessary to configure itself into the target circuit. In our case, since we are focusing on systems comprising a large number of (relatively) simple repeating blocks, we don't need to transfer distinct bit-streams for each of the blocks — we only need to transfer the bit-stream for a **single** block, along with information on how the blocks are arranged. In some cases, this may represent a considerable decrease in the amount of information that needs to be transmitted, depending on the Kolmogorov complexity of the full configuration string [**?**]. There's nothing technically challenging about creating an FPGA that reads a small configuration string and repeatedly applies it to sets of reconfigurable elements inside itself. The challenge comes in deciding on the exact architecture of such a system. More specifically, we need an architecture that is capable of performing *parallel replication* of a large number of blocks. Ideally, **all** the identical blocks of the target circuit would be configured at the same time, thus reducing system configuration time according to the degree of parallelism in the final circuit.

Looked at differently: we are constructing massively-parallel circuits in order to more-efficiently solve problems. But as reconfigurable devices grow larger, we need to consider **configuration time** as part of the total time-to-solution. For example, for a problem that scales very nicely — i.e., one that does not incur dramatically increasing overhead costs as it scales up — the runtime of the system may remain essentially fixed; whereas the config-

uration time will increase with the size (number of elements) in the system. Therefore, the **total** time to solve the problem — the sum of the configuration time and the run time — will still continue to grow as larger problems are tackled. Thus, there may be a total-speedup advantage from parallelizing not only the operation of the target circuit itself, but also from parallelizing its configuration.

So how does one configure a large device with a repeating pattern? Since hardware in intrinsically parallel, the control signals which configure a single block could be sent to multiple blocks, so as to act on multiple regions simultaneously (Figure 3.1). The difficulty with such a scheme is that we don't know *a priori* — when the physical device is being manufactured — what the dimensions of the repeating block will be. The configuration mechanism could be designed to accept a variable width and height, and using a series of multiplexers, it could direct the bits of the bit-stream to the proper (multiple) spots inside the device. Without restrictions on the allowable dimensions of the repeating block, such a general-purpose network would quickly grow quite complex. Moreover, questions of how the configured blocks are to be interconnected need to be addressed, adding further complexity to the reconfigurable device's internal routing network. The possibility of imperfectly-homogeneous designs, i.e., ones where every 1000^{th} block is different, further complicates the design of such a device (but may have a reasonable solution). The challenge here is not really technical *per se*; the challenge is how to design a *single* system that allows this degree of variability.

There is a more-flexible way to achieve parallel configuration, provided the reconfigurable system is *self-configurable*, i.e., its reconfigurable elements are able to reconfigure each other.

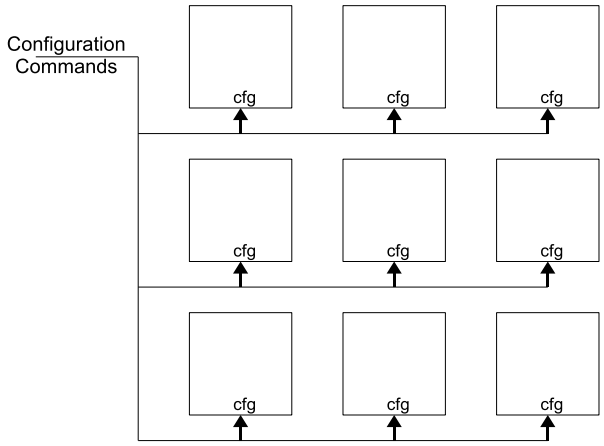

Figure 3.1: General Scheme for Parallel Configuration. A single set of configuration commands are sent to multiple configuration control blocks, to cause multiple regions of the device to be configured simultaneously. This results in a device configuration that consists of a single pattern, repeated throughout the device (in a 3×3 pattern in this case).

If this is the case, then it should be possible to build the **configuration system itself** on-the-fly.

One approach to achieving such on-the-fly configuration of the base configuration layer is to employ a two-layer architecture, i.e., to implement a *meta-architecture* on top of the underlying, hardware-fixed reconfigurable architecture [**?**, **?**]. This meta-layer can then be customized based on how the target circuit is to be configured, so as to employ features such as parallel configuration. In the present work however, we will focus on a *non-hierarchical* meta-layer: if element A can configure element B then B can also configure A.

To use this meta-layer approach, prior to configuring the device to implement the desired tiling of the repeating block, an initial step is performed where we bootstrap the device to implement an appropriate *parallel configuration system* (PCS) inside the device. That parallel configuration system is then given a much shorter bit-stream for configuring a single block, which it then passes in parallel to multiple sets of cells. The advantage to this approach is the flexibility in choosing, at configuration-time, how we want our parallel blocks to be laid out. Our bootstrap of the device thus has two stages:

1. build a custom parallel configuration system inside the device; and then
2. use that system to configure the desired system blocks throughout the device.

It would seem this two-stage bootstrap offers potential benefits in terms of both speed and flexibility. Moreover, it presents an opportunity to deal with the issue of *manufacturing defects*, by having the first stage do some form of defect detection and avoidance, whereby

it will configure the repeating block only on properly-functioning sets of cells. Thus, manufactured devices no longer need to be perfect, but can be allowed some degree of defects, as long as certain system-critical areas are defect-free.

This flexibility does not come for free however: the biggest cost is the increased complexity of the device, since the underlying reconfigurable elements need to be powerful enough to perform configuration operations on each other. With this increase in complexity comes increased system cost and a significant decrease in density.

The biggest issue though may be the increase in overall configuration time: since we are implementing a parallel configuration system in Step 1. above, the size of *that* circuit is going to increase as the size of the final target circuit increases. The question thus arises: **how do we efficiently configure the parallel configuration system itself?**

What we need is a way to efficiently configure the parallel configuration system (PCS). One can expect the PCS to itself contain a high degree of symmetry, and this may itself be a good candidate for parallel configuration. One thus construct *another* PCS (PCS1) whose sole purpose is to configure the PCS. Our bootstrap process now consists of three steps:

1. configure PCS1;

2. use PCS1 to efficiently (in parallel) configure the PCS; and

3. use the PCS to efficiently (in parallel) configure the target circuit.

Here again, the solution seems near-perfect except for the the first step, which somehow

much be performed efficiently (and, perhaps, in the presence of defective elements within the device). of course, we can add another step to the bootstrap: build a configuration system PCS2, use that to efficiently build PCS1, and then proceed as above; and again, we will be faced with a potential bottleneck in the question: "how do we efficiently configure PCS2?"

If this is sounding like a self-referential dilemma, that's because it is! And the solution to this quandary is also self-referential: *we must somehow use the configuration system* ***to configure itself***.

While this may sound nonsensical, it is in fact exactly what a flat (non-hierarchical) self-configurable system is well-suited to. The next section will describe a parallel configuration scheme that is used to configure itself efficiently in parallel. This will be based on a type of circuit called a *Medusa Block*.

3.2 Parallel Self-Configuration: Medusa Blocks

Our goal is to efficiently configure a target circuit. To do so, we will first efficiently configure a Parallel Configuration System (PCS); then we will use that PCS to configure our target circuit in a single set of (parallel) operations. This latter step is straightforward; the entire challenge is in efficiently constructing a PCS. Moreover, we wish to be able to do this on a system that may include defective elements, i.e., cells that cannot be properly configured. Our approach for handling this scenario will be presented a bit further down. For simplicity

though, we begin by describing PCS construction on a *perfect* substrate — one that does not contain any defective cells. First though, we must discuss a construct called an "extensible wire."

3.2.1 Extensible Wires

In a self-configurable system comprised of a uniformly-tiled collection of identical cells (reconfigurable elements), it is possible for cells to be configured so as to implement an *extensible wire*. An extensible wire is a circuit that can pass information — both data and configuration information — from one side to the other, and can use that capability to create an extended version of itself — say a wider version. By repeatedly extending itself, such a circuit can gain access to arbitrary reconfigurable elements within the system. Figure 3.2 shows an example of a simple extensible wire.

The simplest extensible wires, such as the one shown in Figure 3.2, are generally made to grant access to a region of remote cells, and thus usually extend in only one direction. A more-general version of an extensible wire is a circuit called a *Medusa Block*, which is designed to extend in **two or more** directions, and thus grants simultaneous access to multiple regions of cells throughout the system. [1]

The key to efficient parallel fault testing and configuration lies in a particular Medusa Block,

[1] Usage of the term "extensible wire" comes from the fact that, at one time, the only instance of extensible wires were long/skinny circuits designed to access a single remote cell. Though more-general circuits (two- and three-dimensional in their extent) have since been developed, the original nomenclature persists.

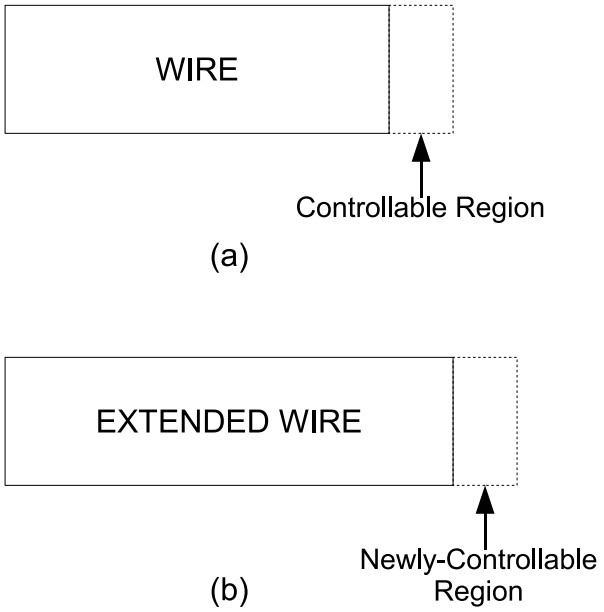

Figure 3.2: Simple Extensible Wire. In (a) the circuit ("wire") is able to configure cells to its immediate right (within the dashed region). In (b), the wire has configured those cells to extend itself, giving itself control over cells further to the right (within the dashed region). The wire can continue to extend its reach further by repeating this process.

with the following characteristics:

- it is designed to extend in two directions (X and Y) along a single plane of a 3-D reconfigurable system;
- it includes circuitry for selectively isolating parts of itself from nearby cells (for the sake of defect avoidance); and
- it can also configure cells in the Z-axis (though not necessarily in a way that allows repeated extension in that direction.

3.2.2 Synthesis of a PCS on a Perfect Substrate

The Parallel Configuration System is comprised of a regular tiling of uniform blocks (each 24×24 cells) in both the X and Y directions (in the $Z = 0$ plane of the 3-D reconfigurable system). Each of these blocks (called a *Medusa Block*) will eventually be responsible for configuring a set of cells *above* it (on the $Z = 1$ plane of the reconfigurable system). Figures 3.3-3.7 show the sequence for synthesis of the PCS. Since the entire PCS resides within a single plane of the reconfigurable substrate, we will for now treat the reconfigurable substrate as being two-dimensional (X and Y axis, with $Z = 0$).

In Figure 3.3, an initial block has been configured inside the system, in the upper-left corner of the reconfigurable substrate. Since this block is relatively small (in contrast to the size of the final circuit), it can be configured using conventional techniques. Note that this initial block is identical to all the Medusa Blocks that comprise the PCS.

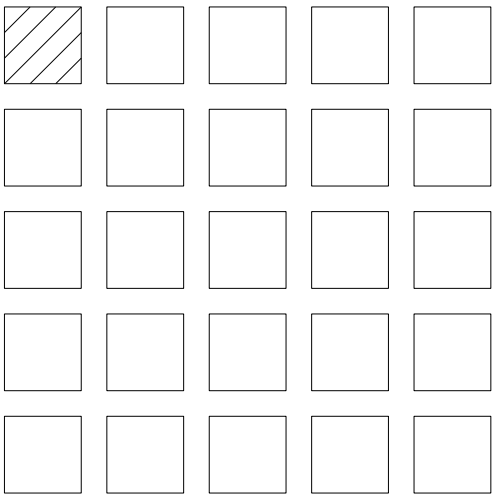

Figure 3.3: Initial Medusa Block, shown hatched in the upper-left corner. From this initial "seed" the much larger PCS can be efficiently built.

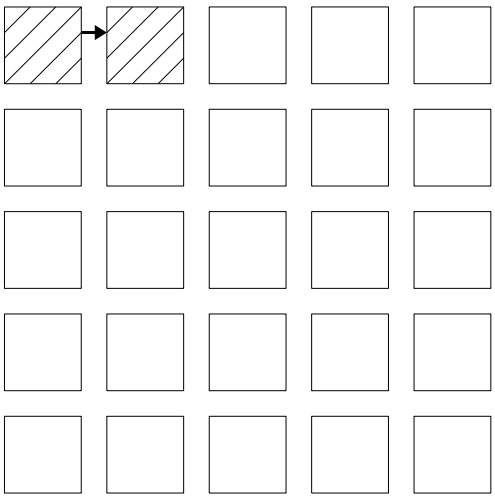

Figure 3.4: Second step of PCS Synthesis: the initial Medusa Block has extended, effectively adding a second block to the east.

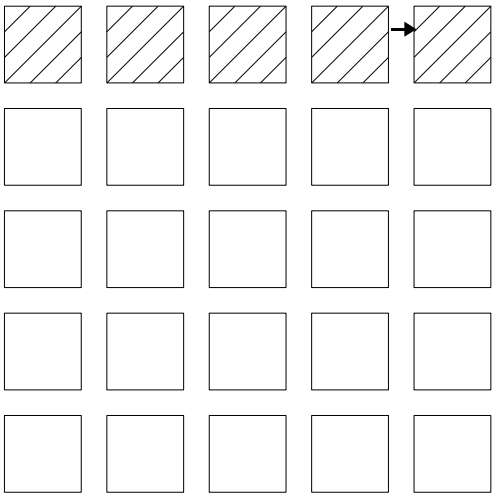

Figure 3.5: After a number of steps, the initial seed has extended across a larger region of the substrate.

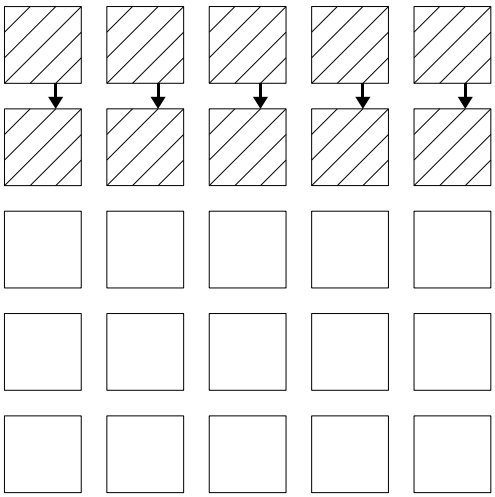

Figure 3.6: The initial row of Medusa Blocks has extended one row to the south. Synthesizing this entire second row requires only a single set of operation: each block in the second row is configured in parallel with the other blocks.

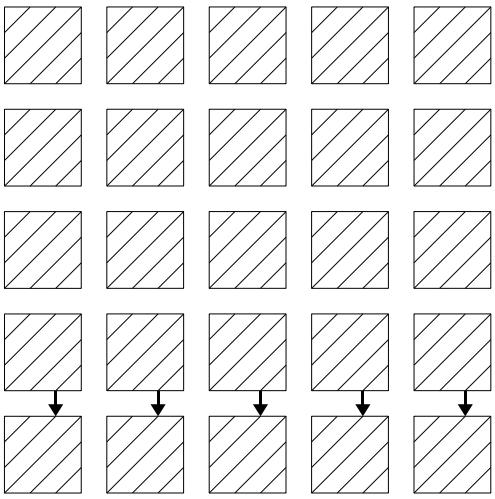

Figure 3.7: Final tiling of the substrate with Medusa Blocks. Each row required only a single set of operations to configure.

The Medusa Block is an extensible wire: it allows cells on its western edge to configure cells on its eastern edge, thereby making it possible to extend itself to the west; and it allows cells on its northern edge to configure cells on its southern edge, thereby making it possible to extend itself to the south. Note that the extension consists of appending identical Medusa Blocks along an edge of the pre-existing circuit. In Figure 3.4, the initial Medusa Block has configured a region adjacent to itself, creating a block that is now twice as wide. This new block configures another region to its right, creating an even wider block. This process repeats until the Medusa Block covers the entire width of the reconfigurable substrate, as shown in Figure 3.5.

This block is now expended in the southern direction. The way it has been built, this expansion occurs in parallel at multiple points along its length. Basically, each copy of the initial block configures an identical copy to its south. Since all these configurations occur simultaneously, the entire circuit doubles its height in a single set of operations. Figure 3.6 shows the Medusa Block after this second row of blocks has been built. This second row now configures a third row (in parallel), and that row configures a fourth, and so on, until the entire substrate has been tiled, as shown in Figure 3.7. This final tiling is one large Medusa Block, comprised of a 2-D tiling of identical Medusa Blocks.

Note that this is not *the* most time-efficient way to configure a two-dimensional region in parallel. For example, blocks could be designed to allow simultaneously configuration of regions to the east and the south. Doing so would mean that, after the initial n steps, roughly $n^2/2$ blocks would be configured, as compared to only n blocks in the current scheme. However,

in terms of order of complexity, the final configuration time is still $O(n)$ for configuring n^2 blocks. The multiplicative constant will be smaller, but at the cost of a much more complex circuit, since multiple channels will be required for conveying multiple configuration strings in parallel. Also, when configuring in multiple directions simultaneously, it becomes possible for a region to be configured from multiple directions at the same time — a condition that must be detected and avoided.

This completes the configuration of the PCS, which is then used to configure the target circuit on the $Z = 1$ plane (details to be discussed below).

In terms of configuration time, suppose it takes t configuration operations ("clock cycles") to build the initial seed. It will then take $2 \times t$ operations to build two blocks, $3 \times t$ for three, and so on. If we wish to make a final tiling m blocks wide, it will require $m \times t$ operations.

To configure the second row though requires only t more operations: so the time for configuring two complete rows is $m \times t + t$, or $(m+1) \times t$. Configuring a third row requires another t operations, and so on. If we wish to make a final tiling n rows tall, it will thus require $(m+n-1) \times t$ configuration operations. After the PCS has been constructed, configuring the target circuit will require an additional t operations, bringing our final count to $(m+n) \times t$.

This is the key to the efficient synthesis of the PCS: if we assume a square region ($n \times n$ blocks), configuration of the target circuit requires only $2n) \times t$ operations. In terms of order of complexity, **configuring n^2 blocks requires only $O(n)$ operations**. This can be a significant speed advantage over a system (such as a traditional FPGA) whose reconfiguration

string length grows linearly with the number of reconfigurable elements, where configuring n^2 blocks requires $O(n^2)$ operations.

3.2.3 Construction of a PCS on an Imperfect Substrate

In practice, it will likely be extremely difficult to manufacture a very large reconfigurable device without defects: insisting on a perfect substrate will simply be cost-prohibitive, as the yield would likely be close to 0. Therefore, we need to have a strategy for operating in the presence of defective regions of the substrate. In fact, we can use the same Medusa Block as described above; we only need to modify the *way* in which we use it.

The modified algorithm is as follows:

1. An initial seed is placed in the upper-left corner of the substrate;
2. The block is used to check regions to the east for defects. This is done by using the control over cells' configuration to assemble test circuits, send test patterns through those circuits, and analyze the outputs for correctness.
3. If there are any defect-free regions to the east, build new Medusa Blocks in them, and return to Step 2.
4. Check regions to the south for defects (similar to Step 2.)
5. If there are any defect-free regions to the south, build new Medusa Blocks in them, and return to Step 2.

Otherwise, the configuration phase terminates.

Figure 3.8-3.17 illustrate this modified algorithm for tiling the substrate with multiple copies of the initial seed. As can be seen, defective regions of the substrate will lead to the creation of "holes" in the final tiling, but will generally **not** result in the elimination of entire rows or columns from the tiling. However, because Medusa Blocks only expend to the south and the east, there are exceptions to this. One exception is if a defect occurs in the leftmost column; this will eliminate all blocks to the south of the defect. Similarly, a defect in the first row will eliminate all blocks to the east of that defect. In general though, lines of blocks "in the shadow" of a defect will be filled in by nearby non-defective blocks.

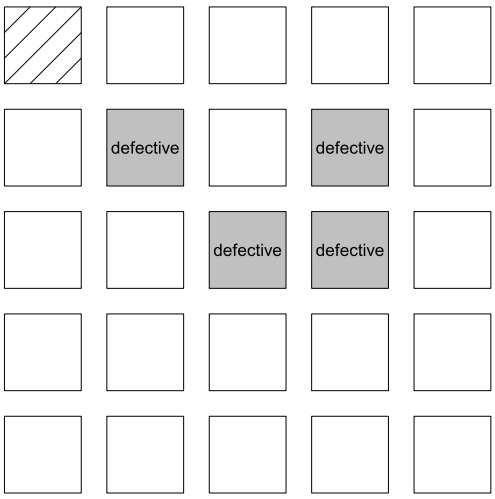

Figure 3.8: Initial seed for synthesis of PCS (in upper-left corner). Grayed-out regions contain defects, and cannot be used in the synthesis.

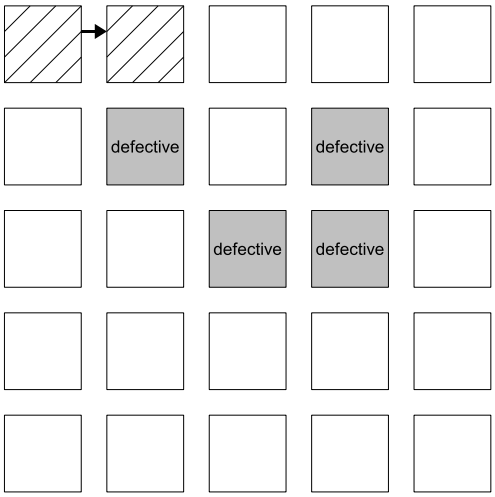

Figure 3.9: The initial seed has configured one block to its right.

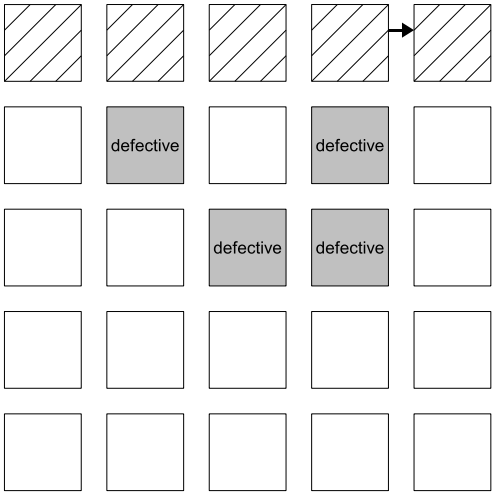

Figure 3.10: Eventually, there will be one entire row of configured blocks.

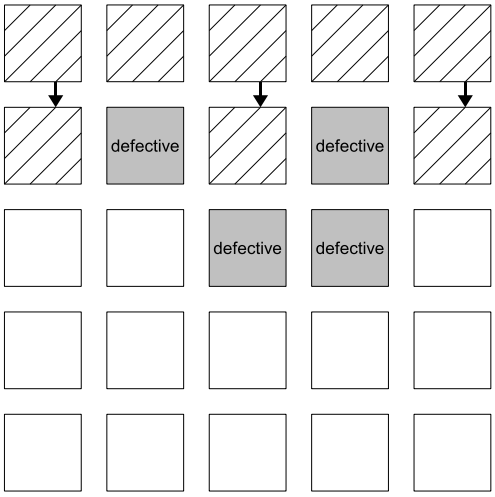

Figure 3.11: Since no more progress can be made to the east, the next step is a single sequence of configurations to the south.

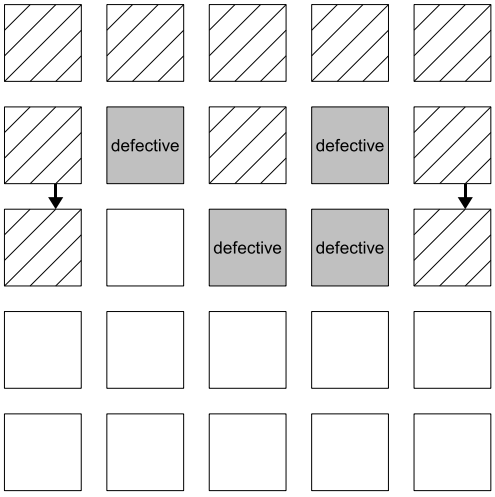

Figure 3.12: Next the system attempts to configure blocks to the east; but this will fail, since all eastern edges are either adjacent to defective regions, or at the easternmost edge of the substrate. The system thus proceeds to configure to the south.

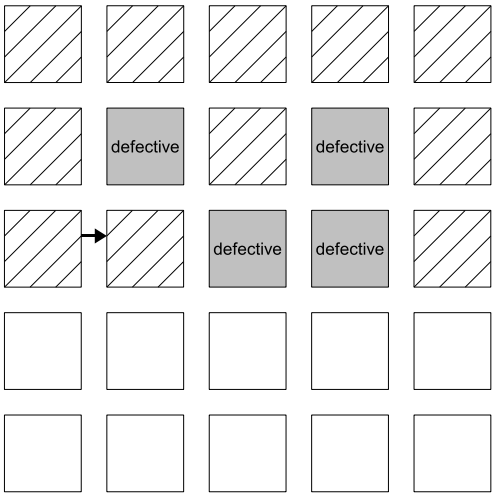

Figure 3.13: Configuration resumes to the east, this time creating one new block.

Using this modified PCS construction algorithm, we can thus achieve a dense tiling of Medusa Blocks on an imperfect substrate, and then use that tiling to create a dense tiling of copies of the target circuit.

3.2.4 Synthesis of Target Circuit

While the Medusa Block (the Parallel Configuration Circuit, whose synthesis is described above) is a two-dimensional structure, each of its blocks is also equipped with circuitry for configuring cells in the Z-axis, i.e., in a layer directly *above* the Medusa Block itself, with each Medusa Block being responsible for configuring the (relatively) small block directly above it. Because each block operates in parallel with all the other blocks, it thus takes a

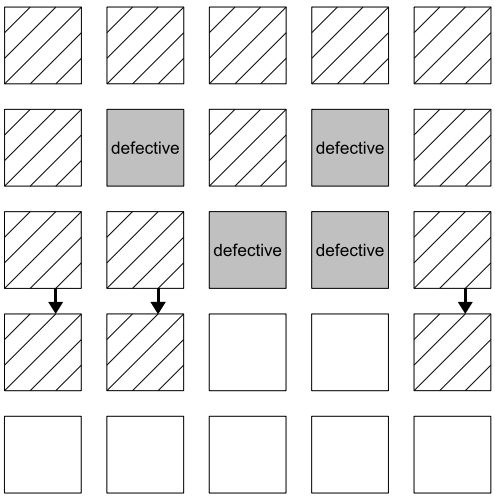

Figure 3.14: Configuration to the south creates three new blocks. For now, the "shadow" of two defective regions has created a hole in the tiling.

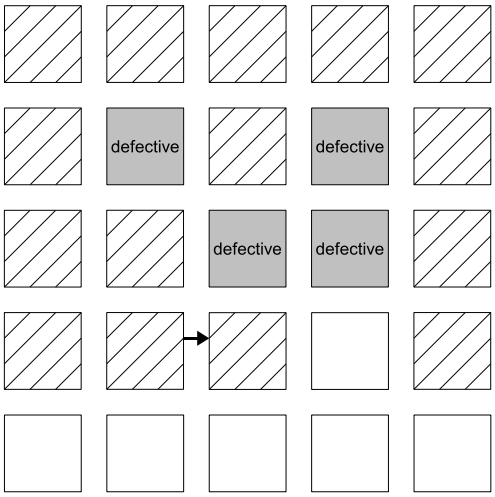

Figure 3.15: Configuration to the east fills in one of the missing blocks.

single set of operations to configure the entire target circuit on the $Z = 1$ plane.

Note that when building the Parallel Configuration System (on the $Z = 0$ plane), it is necessary to check for defects not only on the $Z = 0$ plane, but also on the $Z = 1$ plane above it. This is a simple extension of the testing methodology described above. if there is a defect in a $Z = 1$ region, it is treated the same as a defect in the $Z = 0$ plane, i.e., it terminates expansion in the current direction. Moreover, when a region is considered defective, a nearby block (at $Z = 0$) will set an internal flag that can be detected by the target circuit at $Z = 1$. This flag will allow the target circuit to avoid defective regions, remapping inputs and outputs around them. All of this will be described in greater detail in subsequent sections.

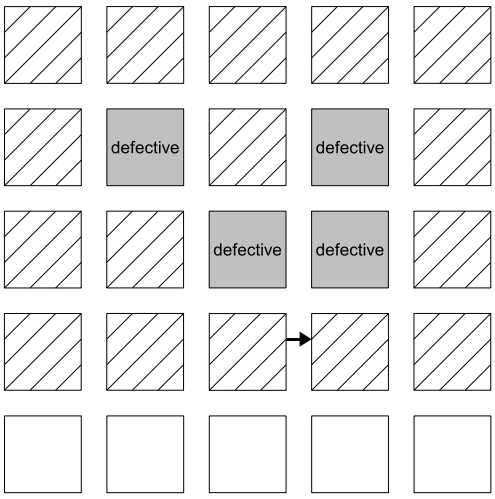

Figure 3.16: A second configuration to the east fills in the remaining hole. Subsequent configurations to the east will fail, so the system again proceeds to perform a southern configuration.

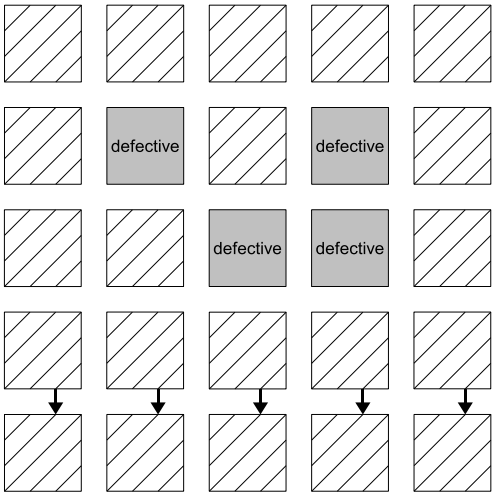

Figure 3.17: Following the southern configurtation, the entire substrate has been tiled. The system will attempt to configure again to the east, and that will fail. It will then attempt to configure to the south, and that will also fail. It is through the presence of this latter failure that the system knows the configuration process is complete.

Figure 3.18 is a movie that illustrates parallel configuration of an array that contains defects. This shows a simulation of an array of reconfigurable elements. Color coding is as follows:

- background-color regions (blue) contain elements whose truth table is filled with all zeros ("uninteresting" elements;
- dark green squares are elements with non-zero truth tables, but whose outputs are all 0 (presently-inactive elements;
- light green squares are asserting at least one D output; and
- red squares are asserting at least one C output, i.e., are configuring one or more neighboring elements.

The three red squares are pre-configured to appear as defective regions, i.e., regions whose elements cannot be successfully configured. Each block in the 5×5 tiling is a single Medusa circuit. Configuration is directed from the upper–left corner, and begins with an initial Medusa block. This block is used to configure a second block to the east, which then configures a third block, and so on, until the entire first row of 5 blocks has been configured. These 5 blocks then, in parallel, each configure a new block to the south (resulting in 4 new Medusa blocks, since one region is unconfigurable). The third row is initially configured with only two Medusa blocks, but a subsequent configuration to the east fill in one of the missing blocks. Likewise, in the fourth row, two configurations to the east fill in the entire row, which then configures a complete row to the south. While the simulation appears to

slow down, in fact the time to configure the final row of 5 elements (in parallel) is the same as the time to configure a single block in the first row. Since the simulation was run on a sequential machine, the more-parallel steps appear to take more time.

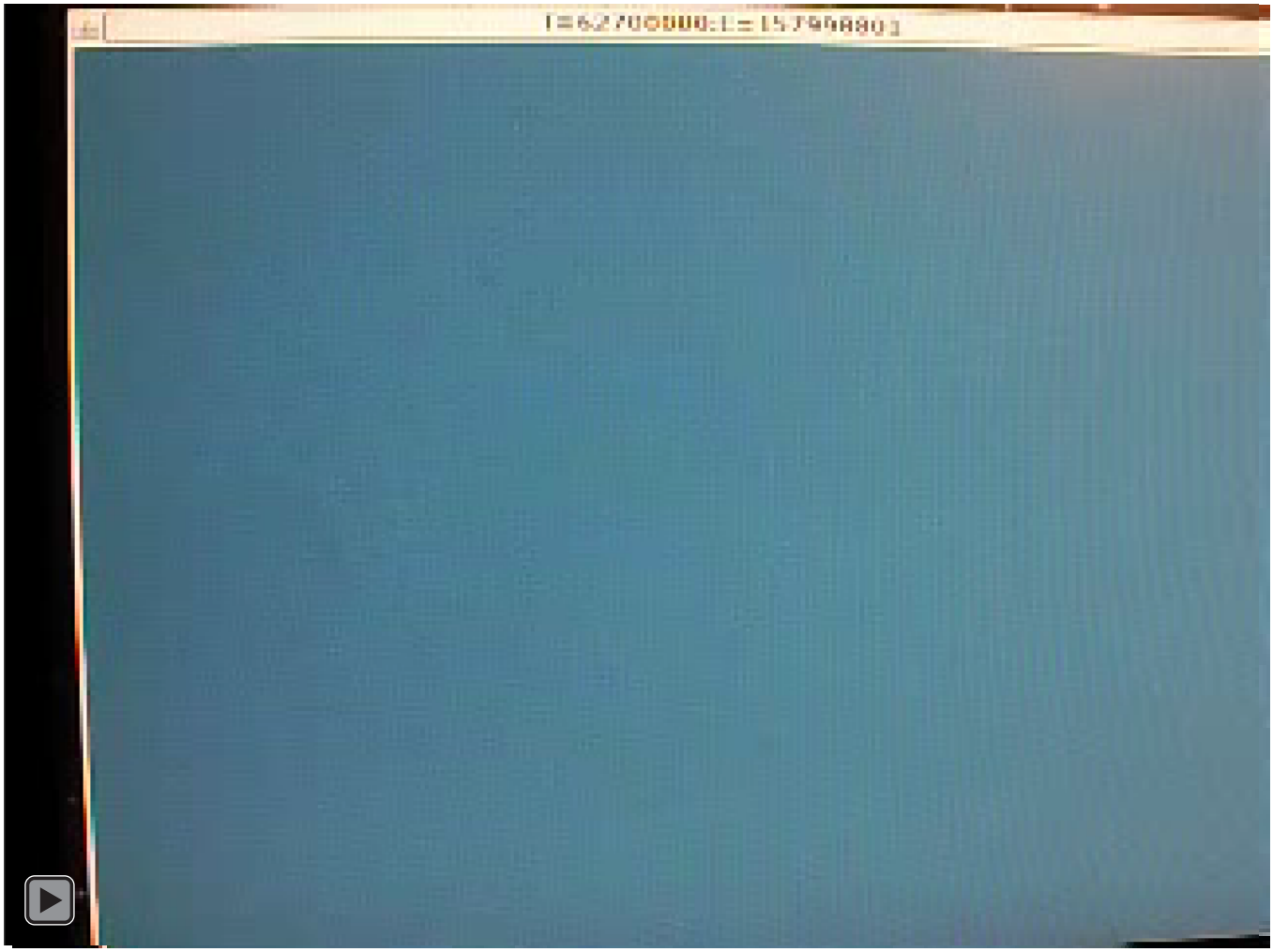

Figure 3.18: Movie showing parallel configuration of an array containing defects. Click on the figure to launch the movie player.

3.3 Detailed Description of Medusa Block

Figure 3.19 shows the layout of a Medusa Block that has the above characteristics. It is comprised of 24×24 cells, and is capable of:

- fault detection in adjacent 24×24 regions to the west or south;
- isolation of itself from adjacent regions to the west or south;
- expansion of itself in the X and Y directions; and
- synthesis of a target circuit in the Z axis.

"Expected Pattern" is a test pattern to be compared to the return value from the Cell under Test (CUT) via the PC line. Comparison takes place when the corresponding Go signal is asserted. Discrepancies result in setting of a Failure flipflop, followed by activation of an Isolation Wall. "We Are Here" is an output signal that informs adjacent blocks that a neighboring block is present. "Are you there" is an input along which a neighbor's "We Are Here" signal is received. "Activate" is asserted at the end of a configuration stage, and unblocks outgoing signals. "Progress" is asserted while Activate is High, and indicates that one or more blocks have been configured since the previous time the Activate signal was toggled. Moreover, for each of these capabilities, the system operates *in parallel* at different points along the perimeter of the Medusa Block. Note that as the Medusa Block expands, the degree of parallelism thus increases.

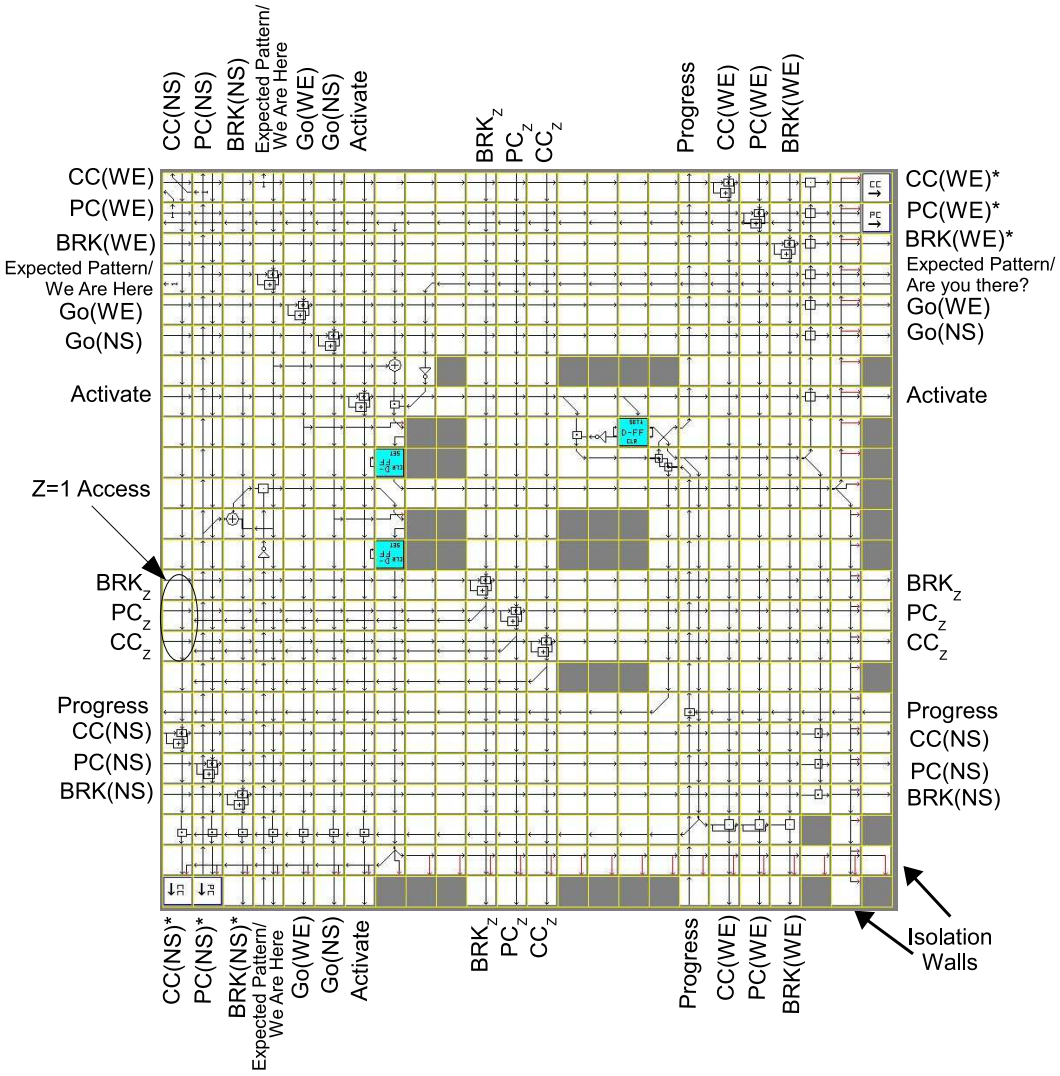

Figure 3.19: 24×24 Medusa Block. Most signals are transmitted from the north and west to the south and east (ORed inside the block). CC, PC and BRK are wire extension commands: CC(WE) is for a west-to-east extension, and so on. Signals marked with a "*" (CC(WE)* on the east for example) are "live," meaning they can configure cells within adjacent unconfigured blocks. Other signals (CC(WE) on the south, for example) are pure data, and will never configure adjacent cells.

3.3.1 Signal Transmission Network

The Medusa Block that is used for parallel configuration is composed of a large number of identical Medusa Blocks, arranged in a regular two-dimensional rectangular tiling. Each of these blocks performs identical functions during the bootstrap of the system: testing of regions for defects; synthesis of new blocks; and eventually, synthesis of the target circuit. To achieve this, identical commands must be transmitted to all blocks, meaning we need a transmission network for carrying this information. This is, in fact, another function of the Medusa Blocks: they collectively form a Signal Transmission Network for broadcasting data and instructions to all blocks.

In a perfect tiling, the design of such a network is simple:

- signals can be injected into the block in the upper-left of the tiling, say into that block's western edge;
- each block in the upper row copies information from the west to the east, and also copies from the west to the south; and
- all blocks in rows other than the first row copy information from the north to the south.

In this way, the signals initially injected into the upper-left corner are transmitted, by the blocks, to every block within the tiling.

One downside to the above scheme is that two types of blocks are required: the blocks in the first row are different from those in any other row. A more-significant downside is that this

scheme does not work well on an imperfect tiling: if a block is missing (due to the presence of a defective cell in the area the block would occupy), then the remainder of the column below the missing block will also be missing. In effect, a missing block "shadows" all the blocks below it. Even worse, if there is a defect in the *first* row, the remainder of that row will be missing, which means all columns below those missing blocks will also be missing.

Fortunately, a simple design change will alleviate both of these problems. Instead of reading information from only the west (first row) or north (all other rows), it is read from both the north and the south, and ORed together. (An assumption here is that unused sides — the northern edge of the top row of blocks and the western edge of the leftmost column of blocks — are receiving values of 0.) Additionally, each block sends the result of that OR to both the eastern and southern edges. Now, even if the tiling of these blocks includes holes, most blocks will still receive the information. The exceptions to this are:

- a bad block in the first row will shadow the remainder of the first row;
- a bad block in the first column will shadow the remainder of the first column; and
- certain patterns of missing blocks will cause other blocks to unnecessarily be missing, as shown in Figure 3.20

In effect, the ORing of signals creates redundancy within the transmission network, providing numerous paths to each block, thus increasing the likelihood that the signals will reach any given block.

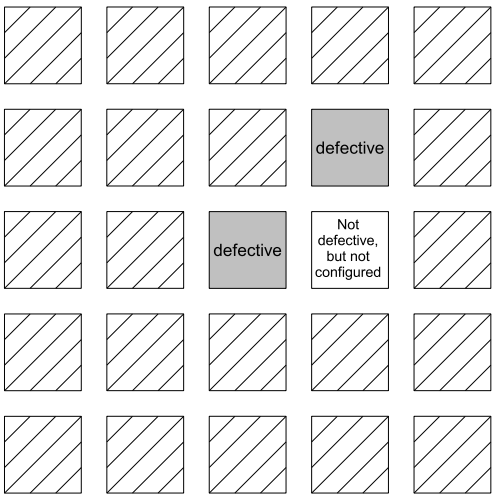

Figure 3.20: A particularly bad arrangement of defects. The region shadowed by both defects will not be configured, even though it contains no defective cells.

This scheme will be used for all signals transmission, with signals sent into the upper-left block and then transmitted to all other blocks. Moreover, when the blocks needs to pass information to something outside themselves, a reverse scheme is employed: blocks OR their own information with information arriving from the south and east, and pass the result to the north and west. In this way, if any block wishes to assert a flag, it may do so, and the upper-left block will receive that set flag.

3.3.2 Extension Mechanism

Extension of the Medusa Block occurs through a small number of signals labeled *CC, PC* and *BRK*. These are the signals for a standard three-channel extensible wire [**?**]. their function can be summarized as follows:

- CC: This signal controls the *mode* of a target cell along the edge of the block. The mode dictates how the target cell responds to incoming its: either as simple data, or as new configuration information.

- PC: This signal supplies a stream of bits to the target cell. The treatment of those bits is controlled by the CC signal.

- BRK: This is an auxiliary line, used to transmit simple data. its specific use is dictated by the circuit being built adjacent to the block. In most cases though, it's used to *break* a wire, i.e., to un-extend or shrink it, returning control to a point prior to the latest extension of the wire.

Because a Medusa Block extends in two dimensions (to the east and to the south), it contains two sets of extension mechanisms (subscripted by "WE" and "NS," respectively). There are thus 6 signals involved in extension: $CC_{WE}, PC_{WE}, BRK_{WE}, CC_{NS}, PC_{NS}$ and BRK_{NS}. As usual, a signal transmission network is employed for distributing these signals to all blocks within the system. However, there is one difference between these extension signals and all other signals in the system. Normally, inputs from the west and the north are ORed together, and re-broadcast to the east and south. The extension signals, however, have a function beyond simple information transfer: they are responsible for actually configuring cells along an edge of one or more Medusa Blocks. So, for the west-to-east signals for example:

- CC, PC and BRK signals are sent into blocks on both the western and northern edges;
- these signals are ORed together;
- the ORed signals are transmitted — *as data only* — through the southern edge of the block; and
- the ORed signals are transmitted — as data only — through the eastern edge of the block into adjacent, already-configured blocks; and
- the ORed signals are transmitted — as *active extension signals* — through the eastern edge of the block into presently-unconfigured regions.

In other words, "WE" signals leaving the south never configure cells to the south; whereas such signals leaving the east will configure cells within previously-unconfigured blocks.

Similar handling occurs for the "NS" signals, which configure cells to the south.

3.3.3 Fault testing

Because we are envisioning working on devices containing trillions (or trillions of trillions) of reconfigurable elements, we expect there to be numerous manufacturing defects in these devices. Since most communication occurs neighbor-to-neighbor, we may also expect that these defects do not impair the entire device, but rather knock out small numbers of reconfigurable elements around each defect. It thus suffices to scan a region for defects prior to configuring cells within that region to implement a Medusa Block.

Note that this ignores the possibility of faults that occur at run-time. As the number of elements in a system grows, the number of failures over any given period will generally increase, leading to a reduced mean-time-between-failures (MTBF). As MTBF drops, rebuilding the entire system in response to every fault becomes less feasible, and a more-local solution is called for [**?**, **?**, **?**]. Numerous possibilities exist for managing such faults, but they are outside the scope of this present work.

Because Medusa Blocks grant configuration control over adjacent cells, we can check for defects as follows:

- extend the block temporarily into an adjacent region;

- construct test circuits in that region;
- stimulate cells and observe their behavior, comparing to expected behavior; and
- note any differences, which are indicative of a defective cell or cells.

It's possible to extensively test every cell within the region under test (RUT) by first extending the Medusa Block to gain control over a cell's inputs and outputs; then configuring that cell with different functions; and then, for each function, sending test patterns to the cell and examining its outputs. For example, a simple function like $OUT = IN$ and a test pattern of 1010... will ensure that neither the inputs nor the outputs are stuck-at-0 or stuck-at-1; configuring a cell with the function $OUT = NOT(IN)$ will ensure that the output is not shorted to the input; and so on.

There is nothing that's very technically challenging about these techniques, but their simulation is time-consuming. Thus, a simplifying assumption was made: that a defective cell will be unconfigurable: its current programming will remain intact, but further attempts to configure it will fail (this is a subtler failure than, say, having a cell only generate fixed outputs). The system then uses the following test sequence:

- configure the cell over which we have immediate access to act as an inverter; send in a pattern of 1010 and make sure the returned pattern is 0101;
- use *that* cell (assuming it's deemed to be good) to test an adjacent cell, using the same inverter/test pattern;

- assuming those two cells are functioning properly, use them to extend the Medusa Block by one cell, and repeat the above steps on the newly-accessible cells;
- continue extending and testing, in one direction, until reaching the end of the RUT;
- use cells near the end of the RUT to change direction, extending the wire south and then west, until reaching the western edge of the RUT;
- use cells near the western edge of the RUT to change direction again, turning south and then again east; and
- repeat this west-to-east and east-to-west pattern, as shown in Figure 3.21, until the every cell within the RUT has been tested.

Assuming all cells have passed their tests, the RUT is considered to be defect-free, and can be used to build a new Medusa Block.

Note that our simple test pattern (input=1010, expected output=0101) will check for basic stuck-at conditions, and will also ensure that the cell being tested is able to be configured (as an inverter in this case).

To facilitate the above test sequence, additional signals are required (see Figure 3.19). The *EXPECTED* signal indicates the bit pattern that is expected to be returned from the cell-under-test (CUT). The returned value is read from the PC line. The GO_{WE} signal indicates that the returned value (from the CUT) should be compared to the value on the EXPECTED line. Comparison occurs at the end of the system clock's *programming cycle*, which is a point

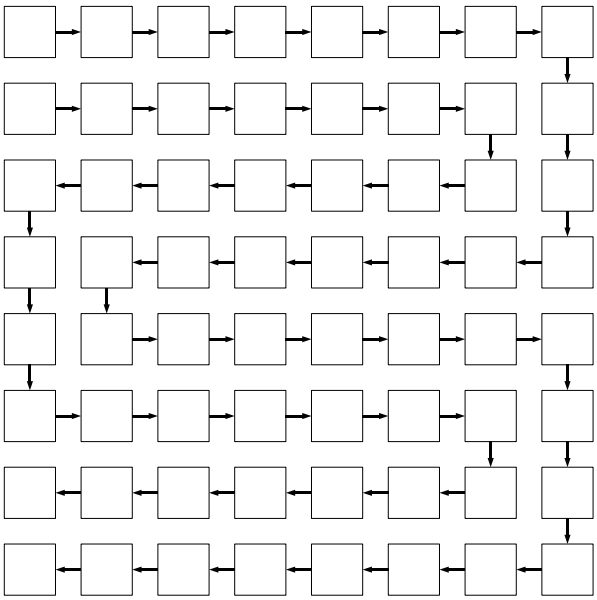

Figure 3.21: Serpentine extension pattern for testing a region of cells. The basic extensible wire is two cells in height, and extends initially from the upper-left corner to the east. When it nears the eastern edge, it makes two clockwise turns, and then extends to the west. At the western edge, it makes two counterclockwise turns, and continues again to the east. This pattern repeats until the entire region has been filled by this single extensible wire. Testing of cells occurs during this extension.

when the system is considered to be quiescent. When the comparison is made, if there is a discrepancy, it is used to load a "1"-bit into a normally-unconfigured cell. Once any 1-bit has been loaded into this cell, it will be read out of the cell shortly thereafter, and will set a *FAILURE* flip flop. This process is irreversible: once a discrepancy has been detected and a 1-bit loaded into the normally-unconfigured cell, the FAILURE flip flop **will** be set shortly after, and will never be unset (unless the entire reconfigurable substrate undergoes a hard, system-wide reset). Thus, if a single test-comparison fails, the FAILURE flip flop will record this event.

The above description covers testing of a region to the east of a Medusa Block. A similar test sequence is also made to the south, in the same way as described above, but triggered by a GO_{NS} signal, and employing a separate FAILURE flip flop for that direction. Thus a Medusa Block can detect and record faults on two separate sides: east and south.

3.3.4 Fault Isolation

If a region of the configurable device contains defective cells, it's most-useful if the entire region (the size of a Medusa Block) is left unconfigured, as opposed to containing a partially-configured block. This is mainly because Medusa Blocks contains structures that indicate to other blocks that they are present, so a partially-built block might mistakenly indicate that it is present, and thus subvert nearby defect-handling circuits.

It's thus desirable to prevent configuration of defect-containing regions. This is accomplished

by a line of cells along the eastern and southern edges of the Medusa Block, in conjunction with the Failure flipflop. When the Failure flipflop is asserted, it causes the corresponding edge cells to become unconfigurable. This not only shuts down further fault testing, but will also prevent the subsequent configuration step where a new Medusa Block would otherwise be built.

The second reason for building these *isolation walls* adjacent to defective regions is to prevent cells within the defective region from affecting the non-defective block. For example, the cell whose output normally indicates the presence of a block could be defective, with its output stuck-at-1, thus incorrectly indicating that a neighbor exists, The isolation wall will prevent the passage of this data value from the defective cell into the non-defective block.

Under the above fault-testing strategy, defects are only sensed to the east and the south. That scheme cannot be directly extended to include regions to the west and north (doing so requires transmission of signals in a north-westerly direction as well as a south-easterly one, and this introduces the possibility of loops, which possess memory). However, a different scheme can be used to test regions to the north and west: Medusa Blocks can be configured to include a digital signature that can be verified by an adjacent block. Since defective regions are never configured, the signature would be missing from a defective region, and this could be used to detect failures (and activate isolation walls) to the west and north. While this is relatively easy to implement, it is not included in the present work — it leaves open the possibility that a stuck-at-1 defect on the edge of a non-configured region could still interfere with the system's operation, though under our assumption that defective cells

are simply unconfigurable, this is not a concern.

Figure 3.22 is a movie that illustrates the automatic detection and isolation of defects within the array. prior to building a new Medusa block in a region, the region is checked for defects. The controlling circuit scans the region under test by building a wire along one edge of the region, turning, building in the opposite direction, and so on. This back-and-forth scan pattern occurs in parallel across all regions to potentially be configured. If any of the tests fail, the entire region is ignored, and a *guard wall* is built between the testing region and the tested region. This shows up as a line of red elements along the edge of a Medusa block. Following this test phase, the regular configuration of new Medusa blocks takes place. Failure also occurs along the eastern– and southern–edges of the array. The simulation is seeded with a single defective element (the yellow element in the lower–right corner of the array). This defective element is detected from both the north and the west, resulting in the building of guard walls on those sides.

3.3.5 Detecting Neighboring Blocks

Because the collection of Medusa Blocks share information among themselves, if any region generates a discrepancy between expected and observed values during the fault testing phase, other adjacent blocks in the same row (for eastern tests; column for southern tests) will also observe this discrepancy. Normally, this would cause blocks to activate their isolation walls adjacent to perfectly-functioning neighboring blocks. Therefore, it's important that each

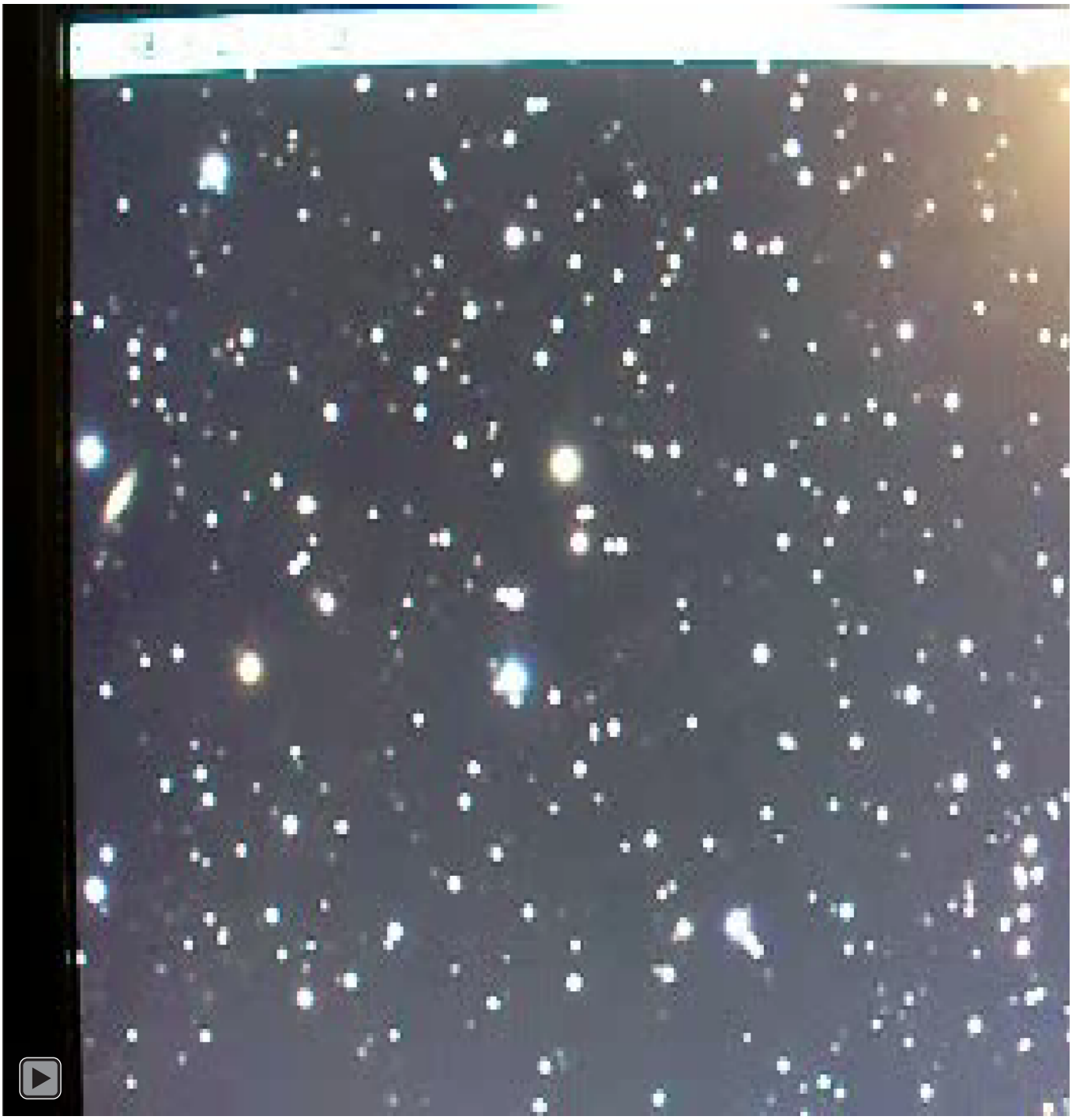

Figure 3.22: Movie showing detection of defective elements. Click on the figure to launch the movie player.

block knows if it is adjacent to a block *that has already been tested, declared defect-free, and configured*. To this end, a "We Are Here" signal is sent out from the western and northern edges of each already-tested-and-configured block, and is read by neighboring blocks on their "Are You There?" inputs. Whenever these inputs are asserted, activation of the Failure flipflop is blocked, thus assuring that once a region has been tested, found to be defect-free, and used to configure a new block, its neighbor will not inadvertently activate its isolation wall.

3.3.6 Activate/Progesss Signal Pair

There is no inherent notion of "compile-time" vs. "run-time." When building a new Medusa Block, each cell in the new circuitry thus becomes active as soon as it is configured. Because of this, it's possible for a partially-configured block to begin configuring a neighboring region before it itself is completely configured. To avoid this, there is an Activation flipflop within each Medusa Block. Until that flipflop is set, the block will remain essentially inactive — in particular, it will not pass configuration commands to any of its neighboring regions.

At the end of each configuration phase, an Activate signal is broadcast to all Medusa Blocks. This signal sets the internal Activate flipflop, which then allows configuration commands to be sent by the newly-activated block.

The Activate flipflop is negative-edge triggered. Therefore, when the Activate signal is high, a block can compare that signal's value to its own Activate flipflop's value. If any block

detects a difference ($Activate = 1$ and $Activateflipflop = 0$), this indicates that the block has just been configured — $Activate = 1$ means the configuration phase is complete, and $Activateflipflop = 0$ means the block has not previously been activated. Any block that determines it is just-now being activated will assert the PROGRESS output signal on its western and northern edges. This signal is a composite of each block's PROGRESS outputs. More-specifically:

- each block receives a PROGRESS signal from the south and east;
- those two inputs are ORed, along with the block's own PROGRESS signal; and
- the resulting signal is output to the north and west.

This means that the upper-left block will output $PROGRESS = 1$ if any of the blocks in the system are themselves asserting their PROGRESS outputs, i.e., if there are any freshly-configured blocks within the system.

As described above, the basic bootstrap scheme is to configure blocks to the east until no progress is made, and then configure to the south once, and then repeat, until the southern configuration yields no progress. The PROGRESS signal is used to guide this sequence.

3.3.7 Configuration of New Blocks

Following the fault-testing phase, defect-free regions are configured to implement new blocks. Configuration of individual blocks is performed via three outputs on the edge of the Medusa

Block, labeled (on the eastern edge, for example) "CC(WE)*," "PC(WE)*" and "BRK(WE)*" (see Figure 3.19). These signals are constituents of a standard extensible wire called a "three-channel wire" [**?**]. Basically, the CC and PC signals are used to control a *target cell's* configuration mode and data input. By properly configuring nearby cells, the CC and PC lines can be used to control increasingly-remote cells, which can themselves be used to further extend this "wire." The BRK signal simply carries an additional bit of information, generally used the "break" the wire, in order to return to controlling a more-adjacent cell.

By using such a wire, it is possible to configure a 2-D region of cells. The process is straightforward: simply build a wire to each cell within the region (one at a time) and configure the cell accordingly. The trick is to do so in a way such that the wires you build later don't re-configure cells you've previously configured. Thus, the order in which the region's cells are configured is critical.

For a rectangular 2-D region, beginning with a wire in (say) the upper-left corner, the procedure is fairly simply:

- the block performing the configuration is extended to the east, to near the end of the region being configured (RBC);
- the block extends to the south, and while extending, configures cells to the east (this configures the rightmost column of the RBC);
- the block backs up one row, and extends to the south, configuring cells to the east (this configures cells in the second rightmost column of the RBC);

- this continues until all but the leftmost three columns have been configured;
- the circuit extends to the south, until it reaches near the bottom of the RBC;
- the cells at the bottom of the RBC's three leftmost columns are configured;
- the circuit backs-up and extends one step less to the south, and configured the cells above those at the bottom of the RBC; and
- this repeats until the entire leftmost three columns of the RBC have been configured.

Figure 3.23 shows the order in which cells within an 8×8 region would be configured under this scheme. Note that the above is for configuration from the west. A different but similar scheme is used for configuration from the north.

3.3.8 Cell Configuration in the $Z = 1$ Plane

Everything that is done on the $Z = 0$ plane of the reconfigurable system is preamble — it is all setup, for the eventual configuration of the target circuit on the $Z = 1$ plane. Most of what has been described above is the process of (efficiently) building a parallel configuration system (PCS) on the $Z = 0$ plane (including testing-for and avoiding defective regions). Once that PCS has been built, it is then used to configure the target circuit at $Z = 1$.

To perform this configuration, a special set of cells are used within the Medusa Block. In Figure 3.19, these cells are located in the leftmost column, next to the signals labeled BRK_Z, PC_Z and CC_Z. These cells work the same as BRK(WE)*, PC(WE)* and CC(WE)*, except

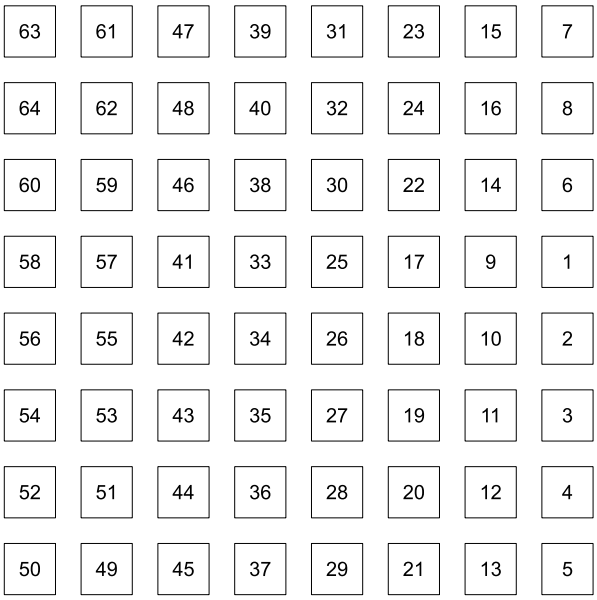

Figure 3.23: Order of cell configuration for an 8×8 region of cells. beginning in the upper-left corner, a wire is built across the top of the region; turned clockwise; and extended from top to bottom. During this extension, regions to the right of the wire's head are configured as desired. Thus, the configuration order is generally from top to bottom, and from right to left (row-major, column-minor). Because the wire used to configure this region is itself three cells wide, the top three rows require a different configuration order, as do the leftmost two columns. This particular order ensures that new cells can be configured without disturbing already-configured cells. The procedure is similar to how one might paint the floor of a room while leaving the paint can in a fixed location near one corner.

that the cell they work collectively to configure is located directly *above* the PC cell (at $Z = 1$), as opposed to the cell in front of it in the $Z = 0$ plane. These cells effectively jump up one plane, and are used to then build extending circuits (wires) on that higher plane, in exactly the same way described above.

Moreover, during the fault-detection phase, these cells are using to test cells on the $Z = 1$ plane. Detection of a defect on that plane is treated the same as defects on the $Z = 0$ plane: the detecting Medusa Block's FAILURE flipflop is activated, and the region containing the defect is thus isolated from the detecting block. This is the simplest strategy for dealing with $Z = 1$ defects: do not configure anything within the region containing the defect. A more-subtle approach would be to place markers — signals generated at specific locations on the $Z = 0$ plane — that could be used by circuits on the $Z = 1$ plane to know that there are nearby defects. Using these markers, the defect-free $Z = 1$ circuits could, for example, route around the defects areas. Some preliminary work has been done in this area.

Configuration of a region is easiest from a corner. While the $Z = 0$ plane is configured over a series of approximately $2 \times n$ steps (for an $n \times n$ region), the plane at $Z = 1$ is configured in a single step. Therefore, it is most important that the configurations at $Z = 0$ be as efficient as possible; inefficient configuration at $Z = 1$ will produce only a minimal negative impact on overall system performance. Thus, the corner locations are used for configuration in the $Z = 0$ plane. This means configuration at $Z = 1$ needs to be driven from a non-corner location. This is less efficient that corner-based configuration, but not actually more-complex. The problem is now similar to painting a floor with the paint can

sitting in a doorway halfway along one wall. Figure 3.24 shows the order in which different sub-regions of a 24×24 region are configured. The order may seem peculiar, but it's simply *one* way to solve the problem; not the only way.

Figure 3.25 is a movie that illustrates three-dimensional parallel configuration. The view is looking down on the $Z = 0$ plane, as a 5×5 tiling of Medusa blocks is constructed. Once all 25 blocks are built, they operate in parallel in construct 25 new blocks on the $Z = 1$ plane. All 25 of these elements are configured at the same time, and construction of all 25 takes the same amount of time as the construction of the first Medusa blocks.

3.4 Simulation

Development and validation of the techniques and circuits described in this chapter were performed using various software tools. Most of these tools existed prior to this research; however, they needed to be extended in a number of ways:

- the Java classes for creating sequences for extensible wires were written strictly for two-dimensional circuits, and thus needed to be extended to three-dimensional circuits;
- the simulator needed enhancement for better display of three-dimensional circuit behavior;
- the ability to inject (and display) synthetic faults at the cellular level needed to be added to the simulation system (this already pre-existed in a less-usable form);

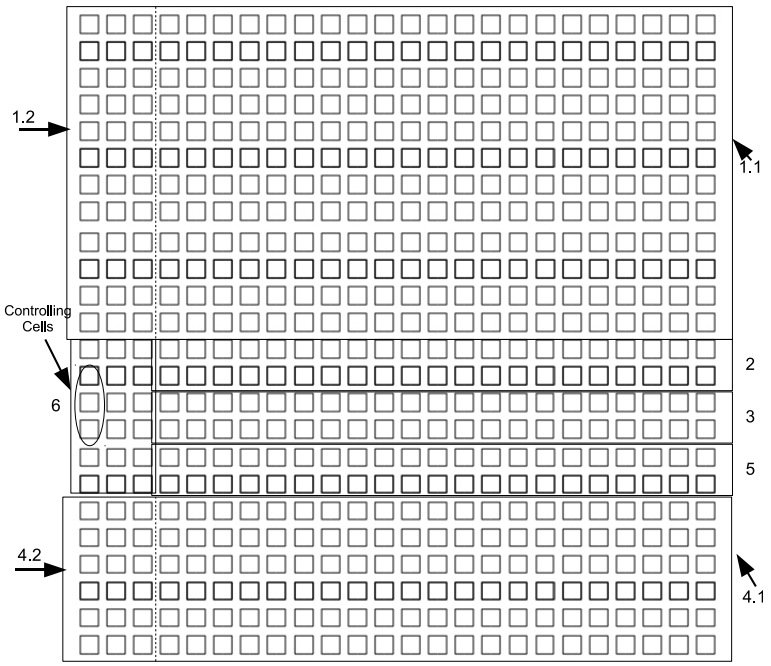

Figure 3.24: Order of cell configuration for a 24×24 region of cells at $Z = 1$. because configuration is being driven from midway along one edge, the configuration order is more complicated than as in Figure 3.23. First the upper part of the region is configured in a normal way. Then two of three thin bands between the upper and lower regions are configured. The lower region is configured next, followed by the thin band in-line with the driving cells. Configuration of a small number of remaining cells in the vicinity of the driving cells completes the configuration of the entire region.

Figure 3.25: Movie showing 3–D parallel configuration. Click on the figure to launch the movie player.

- a way to specify cells that operate in all three axis (the cells that connect the $Z = 0$ and $Z = 1$ planes) needed to be developed; and
- screen-capture tools needed to be created to reduce long simulation times to short replay times.

These modifications represent a significant boost in the capabilities of the development tools for this platform.

3.4.1 Sequences and Supersequences

Control of an extensible wire — say a three-channel wire whose channels are called BRK, PC and CC — boils down to properly generating bit patterns on each of these three channels. Any such particular set of bit patterns may be called a "sequence." Thus, there is a sequence for configuring a target cell as an inverter; there is another sequence for configuring the target cell so as to gain control of a different target cell; and there is another sequence for extending the wire. The first case consists of a single configuration operation; the second requires two operations; and the third requires at least three operations.

The term "sequence" is not rigidly defined: it can refer to any deliberate set of bit patterns delivered to an extensible wire in order to perform desired configuration operations near the head of the wire (including extending the wire). Logically though, sequences may be grouped based on their intended effect. Thus, a set of bitstreams for extending a wire to the east, turning 90 degrees to the south, and then extending to the south while configuring cells to

the east of the wire, might be considered a single sequence.

Note that the term "subsequence" is also sometimes used, to indicate a section of a sequence. Again, a subsequence is still a sequence; the discrimination is logical, based on the part the subsequence plays in a larger-scale sequence/supersequence.

The term "supersequence" is simply a collection of sequences. In the above example, one might use a series of east-to-south sequences, each slightly further to the west, in order to configure a 2-D region of target cells. That collection of sequences could thus be called a supersequence. Of course, a supersequence is still a sequence. The taxonomy is not precise; it's simply a way to logically think about generating bitstreams.

In the operation of the Medusa Blocks described above, we can identify the following sequences:

- fault test to the east;
- fault test to the east at $Z = 1$;
- Medusa Block extension to the east;
- fault test to the south;
- fault test to the south at $Z = 1$;
- Medusa Block extension to the south; and
- configuration of target circuit at $Z = 1$.

The interplay of extension to the east and to the south has been described above, and depends on the results of querying the PROGRESS signal while the ACTIVATE signal is driven high. Moreover, the transition to the final step (configuration of target circuit at $Z = 1$ is initiated based on lack of progress following an extension to the south.

3.4.2 Potential for Pathological Sequences

In theory (and in practice), a small set of elements can be used to reconfigure the entire array. This is, after all, how the array can be initially configured to some desired setup. Without this ability to bootstrap the array, it would likely be of limited use. This capability, however, illustrates a potential failure mode of the system, in which a small set of elements malfunction and begin to change the configuration of other elements within the system. In an extreme case, these malfunctioning elements might reconfigure the entire array, rendering previously-configured circuitry unusable. While such pathological behavior is theoretically possible, in practice it is highly unlikely. When element A (which is perhaps malfunctioning) configures element B, element B is rendered unable to configure any elements itself. For B to subsequently configure another element C, A must first cease configuring B. Similarly, if B begins configuring C, then C is unable to configure any elements itself, unless/until B ceases configuring C. It is thus unlikely that a random malfunction in an element or set of elements – i.e., a malfunction that causes essentially random outputs to be generated – would have the effect of configuring large numbers of other elements. Preliminary experiments with such scenarios reveal that the range of such effects is generally limited to 3-4 elements in any

direction from the malfunctioning element.

Moreover, there are techniques that can be used to build firewalls: rows or columns of elements that prevent malfunctioning elements on one side of the firewall from affecting elements on the other side [**?**].

3.4.3 Keeping Simulation Feasible

Generation of sequences from inside the reconfigurable system, as well as determination of which of several possible sequences to generate based on internal signals (e.g. the PROGRESS signal) is a straightforward task, requiring the development of dense memories to store subsequences, and the implementation of state machines to generate those subsequences in the desired order (possibly based on internal signal values).

While these tasks are simple to perform *in vivo*, their simulation can be expensive in terms of time and memory; moreover, simulating such simple operations at the cellular level serves little benefit: you may get a slightly more-accurate result, but at a tremendous expense in terms of simulation complexity. Because so much of this work is about developing the proper sequences for fault testing, configuration of blocks, and so on, the behavior of sequences themselves are simulated **at the cellular level**. However, the *generation* of such sequences is simulated at a higher level: specifically, sequences' bitstreams are stored in files, and the simulator can read those files and use their contents to set bit values as indicated. This greatly speeds up simulation times.

Chapter 4

Supersequence Generation

Within the context of Cell Matrix programming, a *sequence* is a set of two or more bitstreams, associated with a *multi-channel wire*. Such a wire is comprised of at least two channels: one is called the Program Channel (PC), the second is called the Control Channel (CC). The Program Channel transmits truth table bits corresponding to a cell to be configured. The Control Channel indicates how the PC should be interpreted at the end (head) of the wire:

- if CC=0, then the PC is simply transmitted to the head of the wire (as data);
- if CC=1, then the C input of the wire's head cell is asserted, and the PC is effectively treated as *code*, i.e., it is used to configure the cell at the wire's head.

Through proper manipulation of the PC and CC, one can configure the target cell at the head of a wire; or coerce the target cell to configure another nearby cell; or configure two

more more cells so as to *extend* the wire (or change its direction).

4.1 Block-level Descriptions

4.1.1 Single-Sequence Generator Block

Figure 4.1 shows a block diagram of a *single sequence generator block*. This block generates one instance of a sequence, consisting of a program channel bitstream and one or more control channel streams.

Generation of the block's sequence is initiated by the falling edge of the GO input. Completion is indicated by the falling edge of the EOS output. The PC is generated on the PC output line, and control channel signals appear on the corresponding control channel (CC) outputs (e.g., CC, BRK, etc.).

The entire circuit is driven by a single repeating clock line (CLK). For technical reasons, CLK is not symmetric, but rather consists of 96 0s, followed by 32 1s. The "1" bits are used to trigger generation of the CC bits. Each CC bit is sent during two consecutive ticks of the system clock, with a total of 16 pairs of bits being sent (in serial) out the single CC output. The PC bits are sent out the single PC output (one bit per tick of the system-wide configuration clock).

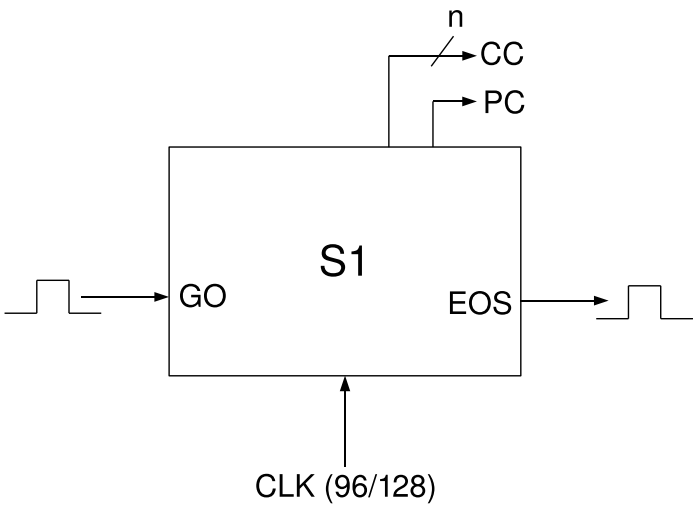

Figure 4.1: Single-Sequence Generator Block Diagram

Sequence S1 is generated on the falling edge of the GO input. PC bits are sent along the PC output, and CC bits are sent on one or more CC outputs. Completion of the sequence is indicated by the falling edge of the EOS output. The entire circuit is clocked with an asymmetric CLK input, consisting of 96 0s followed by 32 1s.

4.1.2 Counter Block

Figure 4.2 shows a block diagram of a sequence counter. The rising edge of the LD input causes the counter to preset to a specified value. The falling edge of the CLOCK input increments the count value by 1. The Z output is asserted when the count reaches 0 and CLOCK is high. NZ is asserted when the count is NOT 0 and CLOCK is high. The "11" in Figure 4.2 indicates that, once preset, the counter block will iterate 11 times between the Z output is finally asserted.

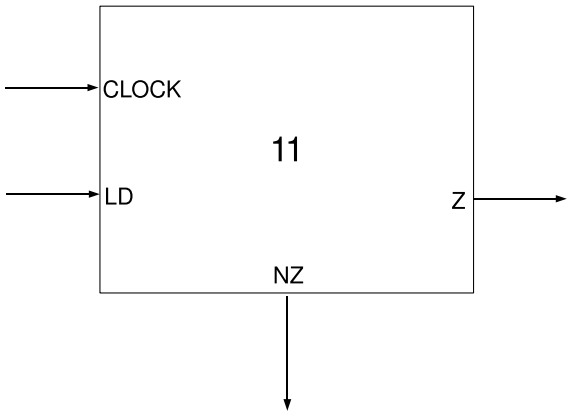

Figure 4.2: Counter Block Diagram

The LD input present the counter, and the CLOCK input increments it. Z and NZ indicate whether the counter is zero or non-zero, respectively. In this example, Z will be asserted following LD and 11 CLOCK pulses.

4.1.3 Supersequence Generation, Block Level

Sequence generators and counters can be combined to produce a supersequence generator. For sequences which are used only once, individual sequence generators such as that shown in Figure 4.1 can simply be cascaded together. Figure 4.3 shows how three such sequences (S1, S2 and S3) could be cascaded. Sequence S1 is initiated by the externally-applied GO signal. S1's EOS signal drives S2's GO, so that as soon as Sequence S1 has been generated, Sequence S2 will begin being generated. Similarly, once S2 finishes, S2's EOS will toggle, which toggles S3's GO signal, thus initiating generation of Sequence S3.

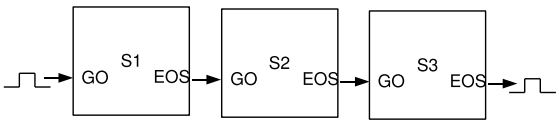

Figure 4.3: Three-Sequence Supersequence Generator

The initial GO signal causes generation of sequence S1. S1's EOS output then initiates generation of Sequence S2, whose EOS signal initiates generation of S3. S3's EOS output indicates the end of the *three-sequence supersequence*. The main clock input, as well as the PC and CC output signals, are not shown in this figure.

Note that the PC outputs from each block should be ORed together, as should the CC outputs. Also, the CLOCK input to each sequence generator should be driven by a single master clock (which can be generated inside the system by using the system-wide configu-

ration clock). These signals are omitted from Figure 4.3.

In general, supersequence generation involves more than simply cascading individual sequences together: it will usually include iterating certain sequences two or more times; conditionally choosing among two or more possible sequences based on an external flag; and perhaps modulating some characteristic of a sequence (such as pre-transferring a guard cell) based on an external condition.

Figure 4.4 shows the basic scheme for sequence iteration. The circuit includes a sequence generator ("S1") and a counter block (labeled "n"). Supersequence generation is initiated by raising the START signal, which drives the counter's LD input, causing the counter to set to its predetermined value. This causes the counter's NZ output to be asserted, which in turn drives S1's GO input, thus arming S1. When the START signal drops, the counter's NZ output also drops, which drops S1's GO signal, thus initiating generation of sequence S1.

When S1 is generating the final step of its sequence, it's EOS signal is asserted. This is connected to the CLOCK input of the counter block, which causes either Z or NZ to be asserted, based on the current counter value. When EOS drops, the Z and NZ outputs also drop, and the counter's value is incremented by 1. Assuming the count was not 0, NZ will have been asserted, which again drives S1's GO input. When NZ drops, GO thus also drops, and S1 restarts its sequence generation.

Eventually, the counter's value will be 0, and the next time S1's EOS is asserted, the counter's Z output will be raised. When S1's EOS drops, the counter's Z output will also drop. Thus,

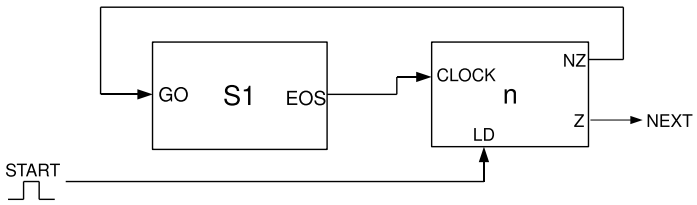

Figure 4.4: Circuit for Generating Multiple Iterations of Sequence S1

Toggling the START signal presets the counter, whose NZ output initiates generation of Sequence S1. S1's EOS increments the counter, which either re-initiates S1 ($count \neq 0$), or drives the NEXT output ($count = 0$, which occurs after n iterations of Sequence S1). Again, the main clock input, as well as the PC and CC output signals, are not shown in this figure.

a rising-then-falling pulse on the NEXT signal indicates completion of the supersequence of repeated S1 generations.

For a w-bit counter which initializes to the value x, the total number of iterations of S1 will be $2^w + 1 - x$ (since the counter value changes *after* the falling edge of the CLOCK input). In general, when a counter block is labeled "n," it means it will iterate n times between the initial LD signal and the generation of the Z output, i.e., the LD signal initializes the counter to $2^w + 1 - n$.

These iteration schemes can be combined to produce more-complex supersequences, as shown in Figure 4.5, which generates the following supersequence:

- sequence S1; followed by
- n23 iterations of a *supersequence* S23; followed by
- sequence S4.

where supersequence S23 is itself composed of;

- n2 iterations of sequence S2; followed by
- sequence S3

The strategy is exactly the same as in Figure 4.4. The START input initiates S1. S1's EOS initializes counter n23, which itself initializes counter n2, which arms sequence S2. S2's EOS increments n2, resulting in n2 iterations of S2 before n2's Z output arms S3. S3's EOS

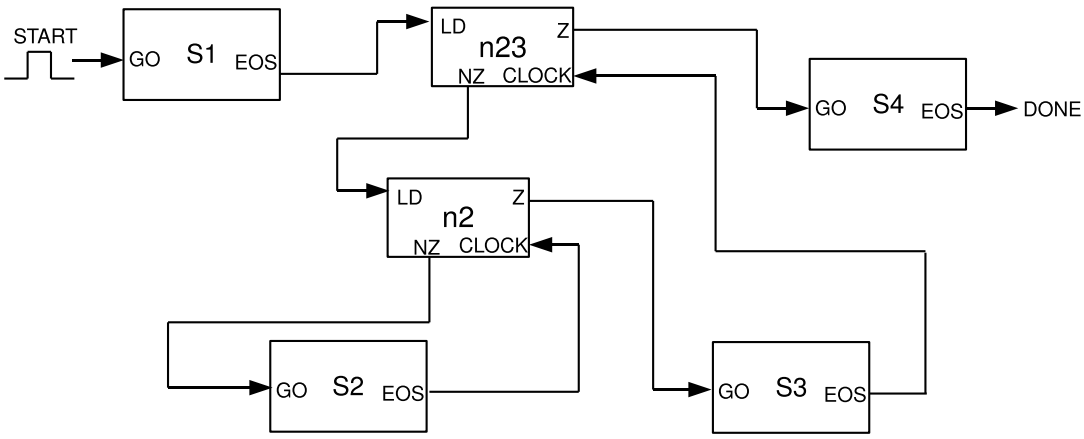

Figure 4.5: More-Complex Supersequence Generator

Supersequence generation is initiated by the START pulse. S1 is generated, followed by $n23$ iterations of the supersequence $[n2 \times S2 + S3]$, followed by sequence S4.

signal increments n23, which causes the supersequence [S2...S2 S3] to be repeated n23 times, followed by a single iteration of sequence S4, followed by generation of the DONE signal.

In this way, fairly-general supersequences can be generated by properly combining sequence generation blocks and counter blocks.

4.2 Supersequence Generator Details

Figure 4.6 shows a complete Supersequence Generator. This circuit is comprised of the following main components (described in detail in the following subsections):

- the PC and CC store and selectors;
- counter blocks;
- the CC shifter and latch;
- timing signal generation; and
- an initial multi-chnnel wire.

4.2.1 PC and CC Store and Selectors

Figure 4.7 shows the subcircuit for generating the PC and CC bitstreams of a single sequence. This subcircuit is composed of a series of shift registers (each composed of three cells: SR0,

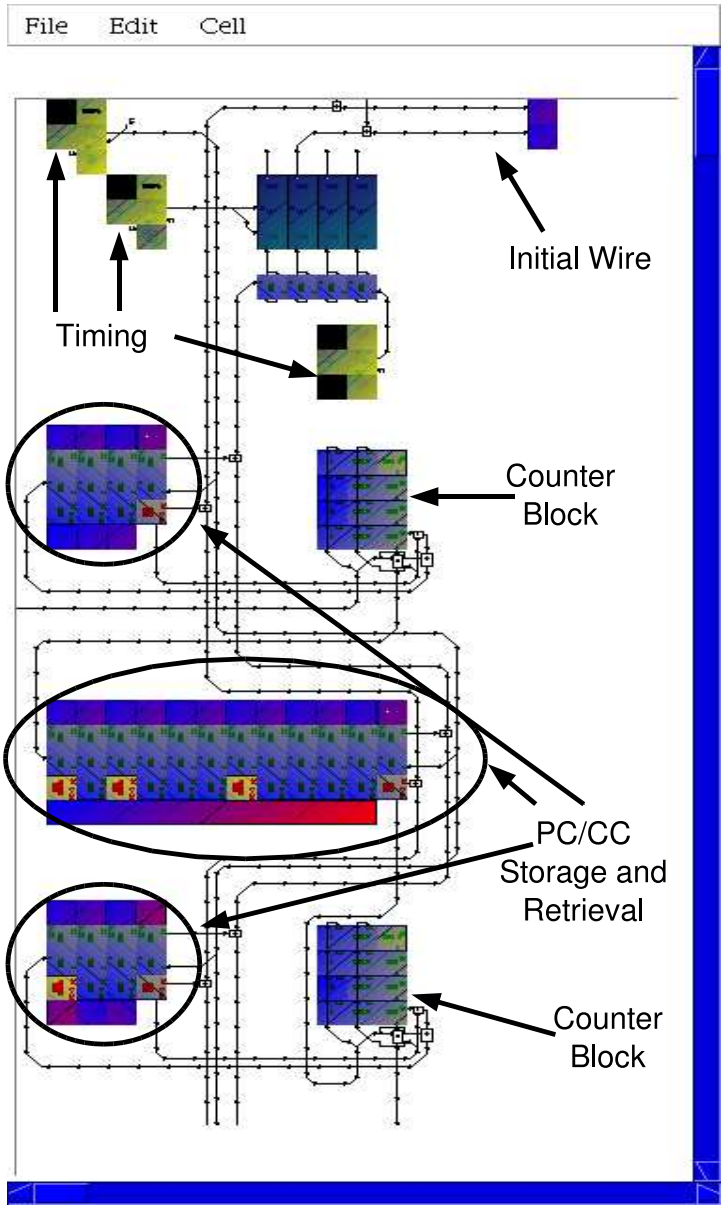

Figure 4.6: Complete Supersequence Generator

Sub-circuits are detailed in Figures 4.7, 4.8 and 4.9

SR1, SR2), which shift in an initial incoming "1" bit from the left, on each rising- and falling-edge of the clock input on the right.

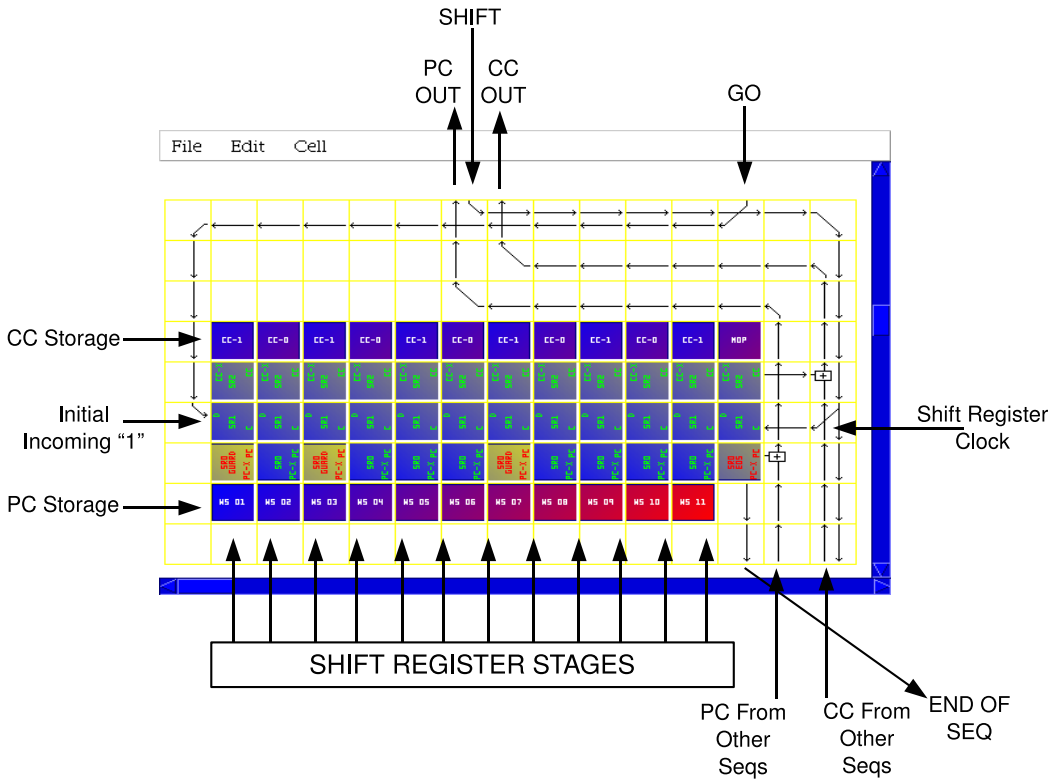

Figure 4.7: PC and CC Storage and Retrieval Subsystem

The GO input supplies an initial "1" which moves through the Shift Register Stages on each downtick of the Shift input. At each stage, a PC and CC is selected, and their bits are sent to PC Out and CC Out, respectively.

The shift register is designed so that, when the clock input C (driven by the SHIFT signal) rises, the D input ("Initial Incoming 1") is pre-shifted into the leftmost stage, and immediately appears at the output (northern edge) of the top cell (SR2). On the falling edge of

C, this value then appears at the output (southern edge) of the *bottom* cell (SR0). These pairs of outputs are used to query stored truth tables to produce the CC and PC bitstreams. Cells to the north of the shift register (i.e., those labeled CC-1, CC-0, CC-1 and EOS) are used to store the CC bitstream, while those to the south (i.e, those labeled S-BLD, CC, PC and NOP) store the PC bitstreams.

As the Initial Incoming 1 is shifted from stage to stage, successive CC and PC storage cells are queried, and their contents are sent to the PC OUT and CC OUT outputs of the sequence generator block. Note that these output lines are ORed with PC and CC output lines from other sequence generators.

For certain steps of the sequence (Steps 0, 2 and 6 in Figure 4.7), the PC Storage cell may be asserting C outputs, which could potentially disturb nearby cells (i.e., the cell is "live"). In these cases, the corresponding shift register's SR0 cell is replaced by one which permanently asserts its C output to the south, i.e., it acts as a Guard Cell for the live PC Storage cell.

Sequences generally end with a single programming cycle during which PC and CC are all 0. The final shift register's SR0 cell is thus a special cell, which does not read a PC Storage cell, but simply generates a signal which indicates END OF SEQUENCE. This signal can be used to trigger further sequence generation.

PC and CC Storage Details

The PC storage scheme is straightforward: each cell's truth table holds the 128 bits which are to be generated on the PC output for that step of the sequence.

The CC storage scheme is a bit more complex. Each CC bit (CC, Break, Transfer, etc.) is held for the duration of an entire 128-tick programming cycle. To take advantage of this fact and reduce storage requirements, a single CC storage cell can be used to store multiple CC signals. This is accomplished by shifting the CC storage cell's truth table bits through a shift register, which latches the bits and presents them to the supersequence generator's multi-channel wire's CC lines.

One further complication is that, under the "standard" truth table bit ordering, the bits shifted out of a C-mode cell at the end of a programming cycle are the bits from the C output columns. This means the CC storage cell would be a "live" cell, the presence of which generally complicate the design of Cell Matrix circuitry. Therefore, the CC storage cell is only placed into C mode near the end of the PC storage cell's programming cycle: specifically, from clock ticks 96–128 (the last 32 ticks). This requires the feeding of an asymmetric clock into the SHIFT input of the sequence generating block. See the section on Timing Control for more details.

Because of this particular clocking scheme for CC storage cells, their truthtable consists of 16 pairs (32 bits) of binary CC values, organized as:

0000 0000 0000 0000 ZZYY XXWW RRBB CCEE 0000.....0000

where the bit pairs are as follows:

- E — EOS (which is not strictly needed)
- C — CC channel
- B — BREAK signal
- R — READ signal
- W — WRITE signal
- X, Y, Z — Reserved, application-specific

The other 8 bit pairs are currently unused.

Such patterns can be created in the libcomp compiler using the .bin directive. For example:

.bin 00000000 00000000 00000000 00001100 0...0

defines a CC storage cell which asserts the C signal. For simplicity, a number of single-bitpair CC cells are defined: one for EE, one for CC, one for BB, and so on. These cells can then be combined in the loader using the OR-PASTE operation. Their icons are also designed so as to make it clear from the OR'd icon which CC bits are set inside the composite cell.

4.2.2 Counter Block

There are many instances where it is desirable to repeat a particular sequence two or more times. In those cases, a *counter block* such as that shown in Figure 4.8 may be used. This

block has the following features:

- an initialization value can be pre-configured in the block, and the count can be reset to that value by asserting a LOAD signal;
- the count can be incremented on each toggle of a CLOCK signal; and
- a pair of outputs — *ZERO* and *NOTZERO* — will indicate whether the current count value is zero or not.

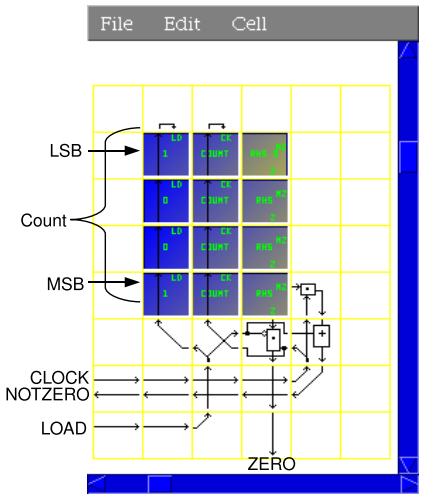

Figure 4.8: Sequence Counter

Count is preset with the LOAD signal, and increments when CLOCK drops from 1 to 0.

ZERO and NOTZERO outputs reflect the counter's state when $LOAD = 1$ or

$CLOCK = 1$.

The initialization value is specified by the cells in the left of each toggle flip-flop (labeled as "Count" in Figure 4.8). The least significant bit is located in the top-most flip flop. For a stored value of n, the counter will count a total of $2^b + 1 - n$ times, where b is the number of bits in the counter's flip flop array. In Figure 4.8, the preset value is 1001 or 9. With 4 bits, this means the total count will be $2^4 + 1 - 9 = 8$. This counter would therefore be used to generate 8 iterations of a particular sequence. Note that, under this arrangement, the count can range anywhere from 2 (when $n = 2^b - 1$) up to $2^b + 1$ (when $n = 0$).

When the CLOCK signal is raised, the ZERO and NOTZERO signals become active, and exactly one of them will be asserted, based on the current value of the counter. When the CLOCK signal drops, the count is increased, and both ZERO and NOTZERO are cleared. NOTZERO is also asserted when the LOAD signal is raised, i.e., when the counter is being initialized.

Using the counter block

The typical usage of the counter block for supersequence generation is as follows:

- the LOAD input is driven by either the End of Sequence (EOS) signal from the previous sequence, or the ZERO signal from the previous sequence's counter block;
- the CLOCK input is driven by the EOS signal of the *current* sequence (the sequence whose generation is being governed by this counter block);

- the NOTZERO output feeds the initial input to the current sequence's PC/CC shift register; and
- the ZERO output feeds the initial input to the next sequence's PC/CC shift register, or the LOAD input to the next sequence's counter block.

So counter initialization occurs at the end of the previous sequence. Following this, the current sequence is generated, and at the end of each generation, the counter is incremented, and the NOTZERO output re-starts the current sequence. When the counter hits a value of 0, the sequence will have been generated the desired number of times, and the ZERO output will begin the next phase of the supersequence.

4.2.3 CC Shifter, Latch, and Timing Control

Because CC bits are held for an entire programming cycle (128 ticks), it's more space-efficient to pack multiple CC information into a single cell's truth table, rather than use separate truth tables for each CC's 1-bit value. This means the stored CC information must be decoded before it is used. Figure 4.9 shows the decoder for the stored CC bits. It consists of a shift register, a series of output latches, and a set of clock generators.

The CC Store Clock generates a "1" signal for the last 32 ticks (ticks 96–128) of the standard 128-tick programming cycle. This signal causes the CC Store to place the current CC storage cell into C-mode, which causes its truth table bits to be sent up the CC Stream line. These bits are fed into the Shift Register, which is clocked by the 1/2 Clock. This means that on

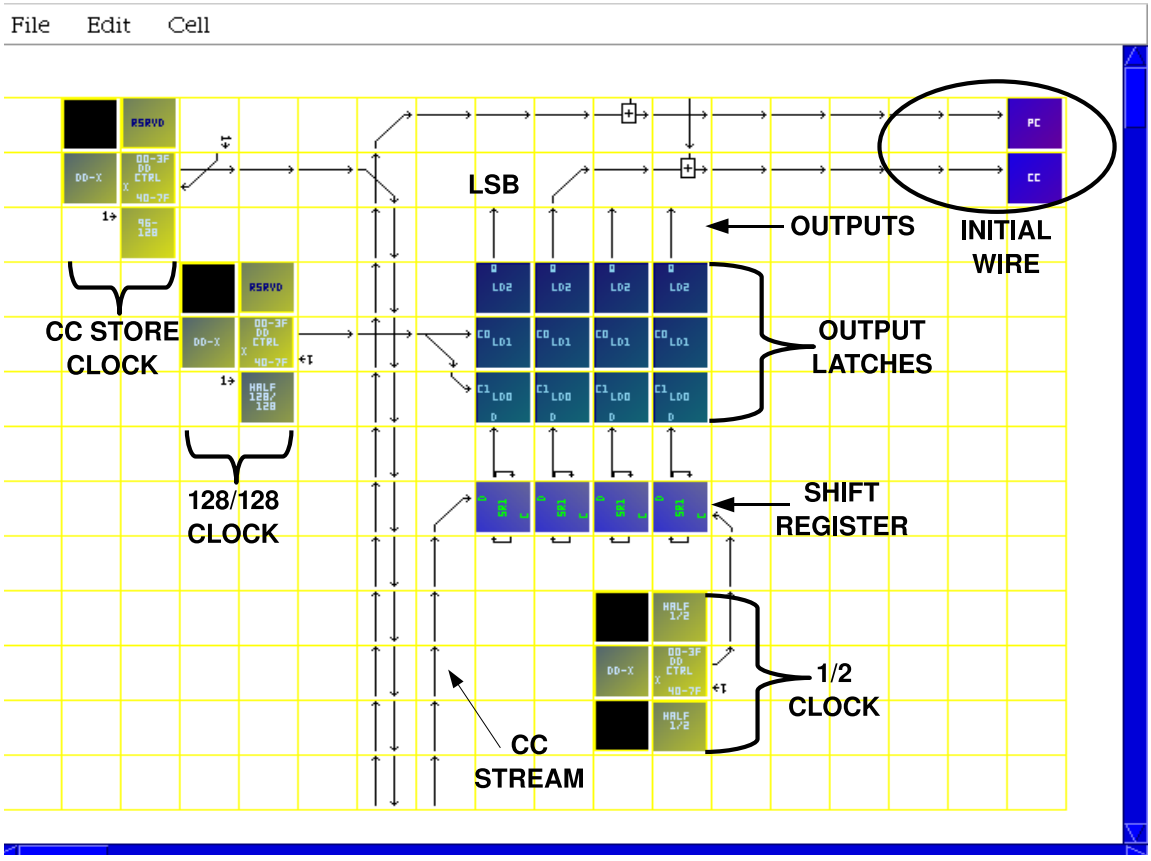

Figure 4.9: CC Shifter, Timing Control and Initial Wire

Incoming CC Stream bits are shifted through the Shift Register, and latched when the 128/128 Clock drops. CC Store Clock generates the Shift input for the PC and CC Storage and Retrieval subcircuit (Figure 4.7).

every other tick of the system clock, a new CC Stream bit is shifted into the head of the Shift Register. Since CC Store bits are stored in pairs, this loads consecutive *distinct* CC Store bits into each shift register stage.

The 128/128 Clock generates a tick on the final (128th) tick of each programming cycle. The rising edge causes the Output Latches to pre-load the outputs of the Shift Register, while the falling edge causes the Output Latches to send those pre-loaded values to the Output lines.

Note that all the clocks in this circuit use the "Double D" clocking circuit, i.e., the desired 128-bit truth table is broken into two sets, with each set stored in the D columns of one of *two* truth tables. Therefore, even if the desired truth table represents a "live cell" (i.e., one which has non-zero C outputs), the generator circuit contains only whose C outputs are all 0.

The combined effect of this circuitry is that, at the beginning of each programming cycle, the Outputs are presenting the CC bits from the previously-selected CC Store. These bits can be sent down the Initial Wire, or used to control other circuitry (e.g., cell storage).

Chapter 5

Sample Application: Solution of the Heat Equation

In this chapter, a sample application — solution of the *heat equation* — is examined, to illustrates the usefulness of a large-scale homogeneous processing system such as the one described above.

5.1 Heat Equation Background

The three-dimensional heat equation [**?**] describes the change in temperature of an object as:

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right)$$

where $u(x, y, z, t)$ is the temperature of a thermal conductor at location (x, y, z) and time t. α is constant, and describes basically how easily heat flows throughout the object (larger α results in quicker temperature changes).

Conceptually, the heat equation tells us that the rate of change in temperature at some point (x, y, z) is affected by how sharply the temperature is changing near that point: larger differences in temperature result in faster temperature changes.

Typically, the heat equation is paired with a set of *initial* and *boundary* conditions, that describe the known state of the system at particular points in time ($t = 0$) and in space (usually points along the edges or surface of the object). These boundary conditions would take the form of $u(x, y, z, 0) = u_0$ (temperature of all points at time $t = 0$ is u_0); $u(0, y, z, t) = u_{x0}$ (temperature along $x = 0$ surface is u_{x0} for all times t); and so on.

For the sake of simulating the behavior of an object subject to certain initial/boundary conditions, it is not necessary to find a closed-form solution to the partial differential equation (PDE) described above. Rather, the system's behavior can be simulated directly from the PDE, using it to repeatedly compute changes in temperature throughout the object; updating the temperature accordingly; and repeating throughout time. For a sufficiently-small timestep (and a sufficiently thermally-smooth object), this technique can yield a good approximation of the system's behavior. This is the basic approach of the *Finite Element Method* [**?**].

5.2 Finite Element Method as Applied to the Heat Equation

Applied to the heat equation, the Finite Element Method requires the region being studied to be divided into a large number of small *elements*, each of which will be assumed to have a constant temperature at all points within it. To update the temperature of an element at (x, y, z), it is only necessary to compute its time derivative $\partial u/\partial t$. Then (for a sufficiently-small Δt), the new temperature can be approximated at time $t + \Delta t$ as $u + \Delta u$, where Δu is approximated as $\Delta t \times \partial u/\partial t$.

It thus suffices to approximate $\partial u/\partial t$; doing so only requires us to know:

- the value of *alpha* (the thermal coefficient of heat flow, assumed to be constant throughout the entire space); and
- the second derivative of u in each of the directions x, y and z.

The former is a constant, and is based on the thermal properties of the object being modeled. The latter can be approximated (for example, in the x direction) by:

$$\frac{\partial^2 u(x,y,z,t)}{\partial x^2} \approx \frac{\frac{\partial u(x,y,z,t)}{\partial x} - \frac{\partial u(x-\Delta x,y,z,t)}{\partial x}}{\Delta x}$$

for sufficiently small Δx.

$\partial u(x, y, z, t)/\partial(x)$ can also be approximated as

$$\frac{\partial u(x,y,z,t)}{\partial x} \approx \frac{u(x+\Delta x,y,z,t)-u(x,y,z,t)}{\Delta x}$$

Similar approximations apply for $\partial u(x,y,z,t)/\partial(y)$ and $\partial u(x,y,z,t)/\partial(z)$.

Combining these gives the following approximation:

$$\frac{\partial^2 u(x,y,z,t)}{\partial x^2} \approx \frac{\frac{u(x+\Delta x,y,z,t)-u(x,y,z,t)}{\Delta x} - \frac{u(x,y,z,t)-u(x-\Delta x,y,z,t)}{\Delta x}}{\Delta x}$$

or

$$\frac{\partial^2 u(x,y,z,t)}{\partial x^2} \approx \frac{u(x+\Delta x,y,z,t)+u(x-\Delta x,y,z,t)-2u(x,y,z,t)}{\Delta x^2}$$

Similarly:

$$\frac{\partial^2 u(x,y,z,t)}{\partial y^2} \approx \frac{u(x,y+\Delta y,z,t)+u(x,y-\Delta y,z,t)-2u(x,y,z,t)}{\Delta y^2}$$
$$\frac{\partial^2 u(x,y,z,t)}{\partial z^2} \approx \frac{u(x,y,z+\Delta zt)+u(x,y,z-\Delta z,t)-2u(x,y,z,t)}{\Delta z^2}$$

This matches the intuitive understanding of heat flow: the change in termperature at a given point is a simple function of the thermal gradient in the $+/-x$, $+/-y$ and $+/-z$ directions near that point.

5.3 Modelling Thermal State in a 2-D Region

To model heatflow across a two-dimensional (flat) object, the surface being modeled is first divided into a set of small elements. Since this system is being modeled on a reconfigurable

device composed of a large number of reconfigurable cells, a collections of cells can be used to implement each of the heat equation elements; and since the cells are layed out in a regular 2-D tiling, it's natural to map the surface being modeled to the physical layout of the reconfigurable system, as show in Figure 5.1. In this example, the object being modeled is divided into nine elements, and each of those elements is individually modeled by a 4×4 set of cells (in practice, many more elements would be used to model the surface; and the number of cells required to model a single element would be much higher).

Note that such a mapping of a problem space to a set of spatially-distributed elements is similar to what is sometimes done in using cellular automata [**?**] to solve such problems. While the mapping process may be similar, the fundamental computing method is quite different though. With cellular automata, there is generally a global ruleset defining the state of each element based on its current state and the state of its neighbors; transitions occur in unison based on a system-wide transition clock; and implementing specific functions (adders, multiplexers, etc.) requires intimate knowledge of the automaton. With the self-configurable architecture described herein, the state of each element depends not only on its neighbors' states, but on a per-element ruleset for mapping inputs to outputs; transitions in outputs occur more or less immediately (based on the speed of the elements' internal logic), without regard for the transition of any other elements in the system; and implementing blocks such as adders or multiplexers only requires knowing the truth table of such blocks.

To model our PDE on such a system, the element at coordinates $[c, r]$ (corresponding to one element in the sub-divided object being modeled) stores its current temperature $u(c, r, t)$.

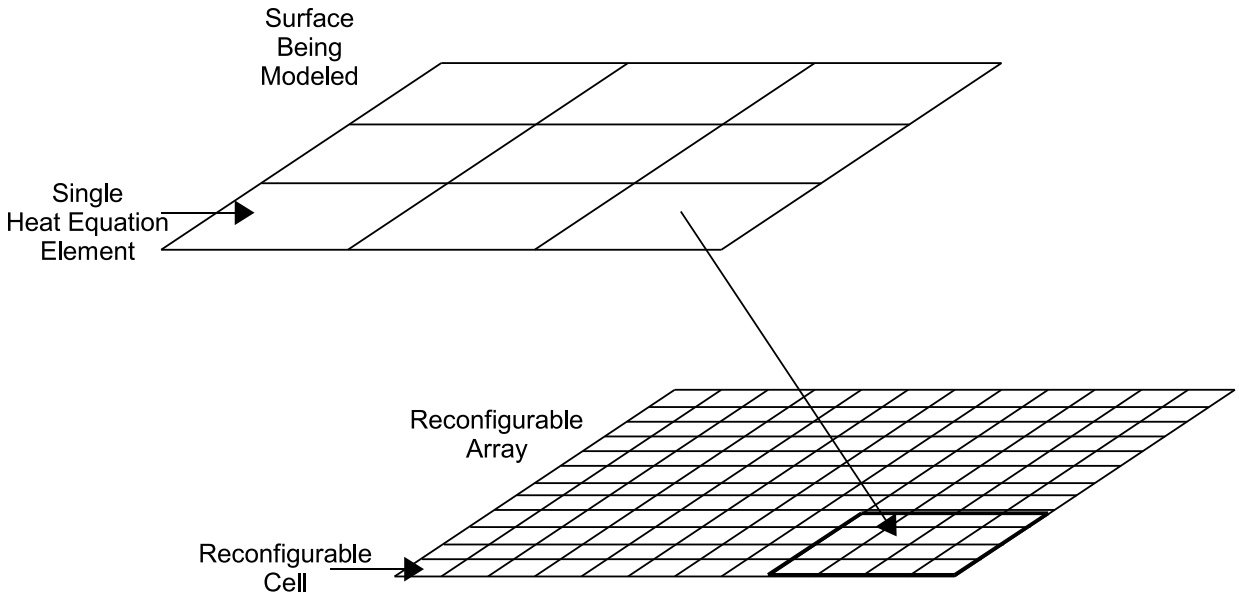

Figure 5.1: Modeling of Heat Flow across a 2-D Surface. The surface being modeled is divided into a number of elements, each of which will be treated as having a uniform temperature across its entire surface. The PDEs that describe thermal flow are approximated by difference equations, and each element's thermal flow is modeled on a series of reconfigurable cells in an underlying reconfigurable substrate. In this example, the tiling of the surface is uniform, but in practice an *irregular mesh* might be used to more-accurately model the system where surface variation is higher.

The distance from one element to another in the x axis is taken to be Δx, and similarly in the y axis. Thus, $u(c + \Delta x, r, t)$ is the same as $u(c + 1, r, t)$, the temperature of the neighboring element to the right (east).

Under this mapping, the difference equations for the change in u has the form:

$$\frac{\Delta u}{\Delta t} = \alpha(\frac{u(c+1,r,t) + u(c-1,r,t) - 2u(c,r,t)}{\Delta x^2} + \frac{u(c,r+1,t) + u(c,r-1,t) - 2u(c,r,t)}{\Delta y^2})$$

Since Δx and Δy are the distances between points represented by adjacent elements, these can be treated as being equal; and, moreover, by properly scaling α, one may set $\Delta x = \Delta y = 1$. The difference equation is thus:

$$\Delta u = \alpha \Delta t(u(c+1,r,t) + u(c-1,r,t) + u(c,r+1,t) + u(c,r-1,t) - 4u(c,r,t))$$

If time is scaled so that $\Delta t = 1$, then the equation for updating the temperature of any element $u(c, r, t)$ is very simple:

$$u(c,r,t+1) = u(c,r,t) + \alpha(u(c+1,r,t) + u(c-1,r,t) + u(c,r+1,t) + u(c,r-1,t) - 4u(c,r,t))$$

This update thus only requires that the current temperature of the given element and its four nearest neighbors. A circuit can therefore be designed for implementing each element, such that the circuit will:

- store that element's current temperature;

- read the temperature of each neighboring element (so as to allow calculation of its next thermal state);
- transfer its current temperature to its immediately-neighboring elements (so they may calculate their next thermal state);
- compute its next thermal state using this temperature information, along with its pre-stored value of α; and
- update its own temperature (in unison with neighboring elements' own updates).

5.4 Circuit/Cell-Level Description of 2-D Heat Equation Solver

Figure 5.2 shows a block-level diagram of a basic element in the heat equation-solving system. This circuit is a straightforward implementation of the difference equation describing temperature change (above). The circuit operates as follows:

- it reads temperature (N) from the neighbor to the north;
- it multiplies its own temperature (u) by 4 (a simple left-shift of two bits);
- it subtracts $4u$ from N;
- it adds the temperature from the east (E), south (S) and west (W) to the above difference;

- it multiplies the resulting sum by α; and
- in synchronization with a global clock, it adds the final sum to its own temperature u.

For conducting cell-level experiments on this system, temperature is treated as an 8-bit signed integer: -128 is the coldest temperature, and +127 is the hottest. Parallel transfer paths are created between each element and its four immediate neighbors. By setting $\alpha = 0.125$, one can right-shift three bits to multiply a quantity by α.

10-bit adders and subtractors are used to compute $N + S + W + E - 4u$; the result is right-shifted three bits; and that result it added to u. u itself is stored in a master/slave flip-flop, so the updated value of u can be computed without affecting the current value of u prematurely. Update of the register containing u is made in a two-phased process (preload/load).

Figure 5.3 shows the cell-level circuitry for implementing a single heat-equation-solver element, and is a fairly straightforward implementation of the circuit shown in Figure 5.2. The element's temperature is stored in a master/slave flip flop: its input is latched on the rising edge of its Clock input, and the output is changed accordingly on the falling edge. In this particular implementation, there is a single cell who's internal truth table is used to generate a repeating bit pattern, by having that cell placed in configuration mode. This produces a free-running bit pattern, that is used to step the simulation.

The complete circuit for a single element is 64×64 cells and, as is often the case with large circuits on this substrate, much of the space is consumed by simple routing of signals.

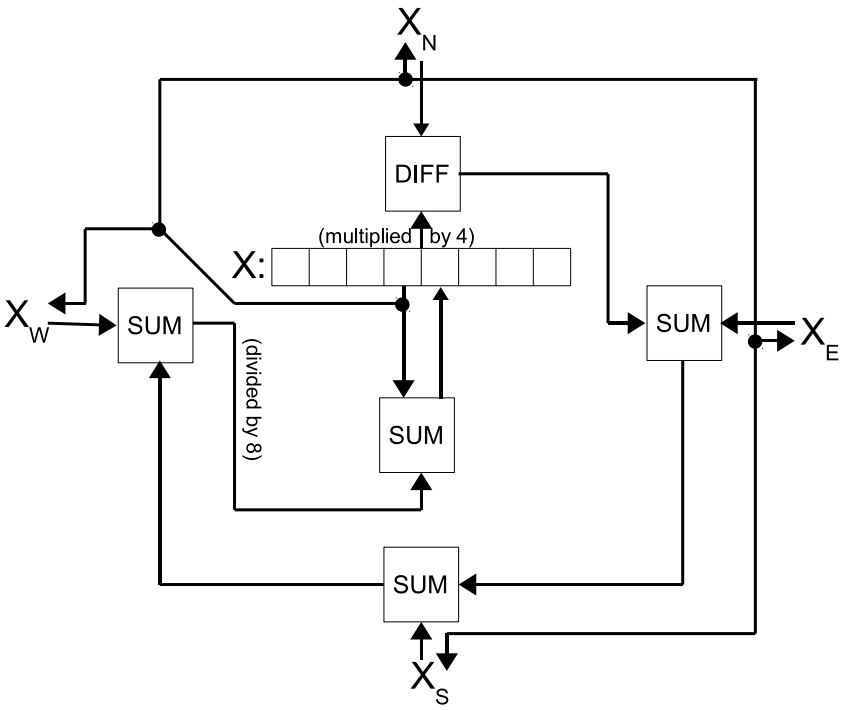

Figure 5.2: Block Diagram of a 2-D Heat Equation Solver. A number of these elements are tiled in a 2-D array, with each internal element connected to four neighboring elements. Temperature is represented as an 8-bit signed integer, and is stored inside each element in the register labeled "X." Each element sends its own temperature to its neighboring elements, and receives each neighbor's temperature through an input along its perimeter (elements along the edge of the tiled array read *boundary conditions* from one or more of their inputs). X is multiplied by 4 (via a simple 2-bit left shift); the four input (X_N, X_S, X_W and X_E) are summed, and $4x$ is subtracted from that sum. The difference is divided by 8 (right-shifted 3 bits): this corresponds to scaling the difference by a thermal coefficient $\alpha = 0.125$. The scaled difference is added to X to compute the element's temperature at the next time step. X is stored in a master/slave flipflop, so that changes to its input are latched before the output changes. This allows all elements to update their temperature simultaneously without any critical races.

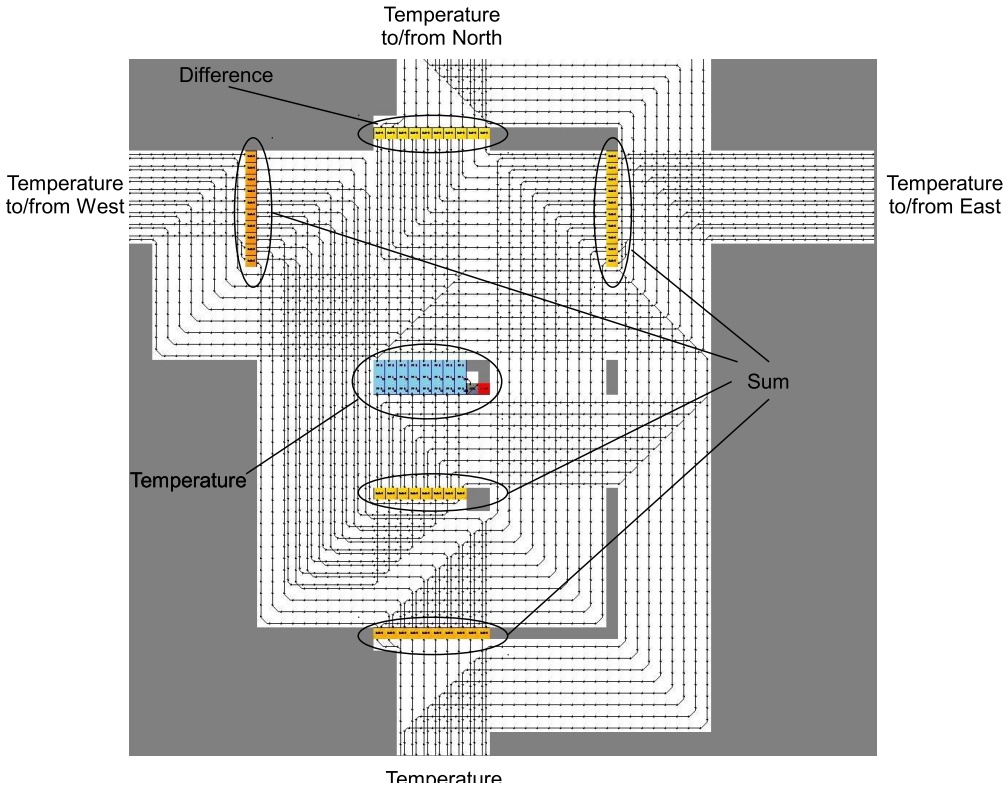

Figure 5.3: Cell-Level Implementation of a Single Heat Equation Element. The element's temperature is stored in the register in the middle of the circuit, and is sent to each of the element's neighbors (on the north, south, west and east. The Temperature is shifted left 2 bits, ot produce $4 \times Temperature$, which is subtracted from the temperature received from the north. To this difference is added the temperatures from the east, south and west. This final sub is shifted right 3 bits, and added to the current temperature. The final result is loaded into the Temperature register on the rising edge of its clock input. On the falling edge of the clock input, the register's outputs are changed to reflect the latched input.

5.5 Simulation and Results

For the initial tests of this circuit, a square region was divided into 64 elements (8×8). Since each element is itself composed of 64×64 cells, the final circuit is 512×512 cells (262,144 cells). It is difficult to observe that many cells and make sense out of their behavior in the regular simulation tool. Therefore, an augmented toolset was developed, based on a client/server model.

The simulator itself was enhanced to operate in a *server* mode. In this mode (which is triggered by a command-line switch), the stdin and stdout channels of the simulator are connected to sockets instead of to the controlling terminal. This allows a client to control the simulator (which is normally command-line based, i.e., it responds to typed commands) by sending commands through a socket; and to monitor the behavior of the simulator by requesting various outputs (cell state, truth table dumps, output values, etc.) and reading them from another socket. The simulator was also enhanced to support a "-dark" switch, which suppresses all graphical output. This allows the simulator to run on a true server mode, without requiring any sort of terminal I/O on the CPU, and to run more-quickly by avoiding screen updates. Figure 5.4 shows the basic client/server model for this system.

In addition to the enhanced simulator, a client program specific to simulation of the heat equation was developed. This client can perform the following actions:

- display a graphical image of the simulated 2-D plate whose thermal behavior is being modeled;

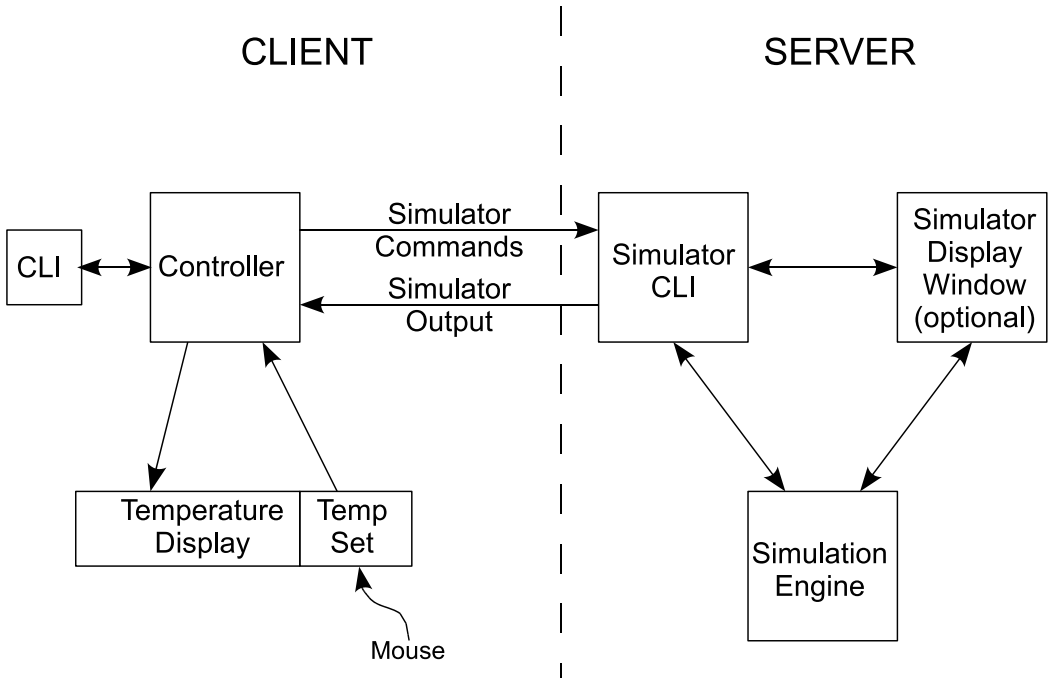

Figure 5.4: Client/Server Model of Heat Equation Simulation. The server side is comprised of the usual simulation elements: the command line interface (CLI); the underlying low-level simulator itself; and a graphical display of cell activity (which can be completely disabled). The server is responsible for cell-level modeling the behavior of the heat equation elements. The client side communicates with the server through a pair of sockets. Commands can be entered via its own CLI. Once the controller has been put in "free-running mode," it repeatedly requests the temperature of each element, and updates its temperature display. Additionally, a mouse can be used to set the temperature of boundary elements, and these *boundary conditions* are repeatedly sent to the server, which sets edge-cell inputs accordingly.

- establish a connection to a server-mode simulator (since TCP/IP sockets are used, the client and server do not need to be running on the same machine — they can be anywhere on the internet);
- send initialization commands to the server, to load the heat equation element circuitry; create input/output vectors; set up other simulation parameters; and initialize the display if the simulator is not running in -dark mode; and
- enter a main *free-running simulation* mode.

In the free-running simulation mode, the mouse can be used to increase or decrease the temperature of any boundary elements (by pointing to the element and rolling the scrollwheel up or down). The temperature of boundary elements is set by setting the inputs to boundary elements on sides where there is no adjacent element; i.e., for an element on the northern edge of the region being simulated, that element's northern inputs are not connected to a northern neighbor. They are, instead, available to be set at will, based on the boundary conditions at that point. Whenever a boundary condition is changed, a "set vector value" command is sent to the simulator to effect the requested change.

The client sends a *step* command to the simulator, which causes the system to advance state to the next timestep. When the simulator has stabilized, the client reads the temperature of each element (by reading the output of each flip flop from which an element's temperature-storage register is composed); converts that temperature to a color (blue being the coldest, red the hottest); and updates its graphical display of the simulated material. Requested

boundary value changes are sent to the simulator, and the process repeats. In this way, one can interactively adjust the temperature of the simulated material, and observe graphically the change in temperature over time.

Because the client communicates with the server over a pair of TCP/IP sockets, it can also interact with multiple servers, i.e., with a distributed simulation being performed across multiple nodes of a cluster. This approach will be used in the *Simulation* chapter, where multiple nodes are used to distribute the load of simulating a larger region's thermal behavior.

Figure 5.5 is a movie that shows the behavior of the client–server–based simulation of the heat equation system. The region on the left is a view of the element-level simulator. Each block in the 8×8 tiling is itself a 64×64 circuit for updating the temperature of a single mesh element. The window in the upper-right region shows the surface across which heat flow is being simulated. Temperature is color–coded (red=hot, blue=cold). Boundary temperature can be selected with the mouse, and these boundary conditions are supplied to the unconnected inputs of mesh element circuits that are situated along edges. As boundary temperatures are changed (by pointing the mouse to a boundary element and using the scroll wheel), the change in input values can be seen to propagate throughout the 512×512 circuit on the left. At the same time, the temperature of mesh elements in the surface being modelled (upper–right) can be seen to change, as indicated by a change in the elements' color.

Figure 5.5: Movie showing behavior of heat-flow-simulation circuitry. Click on the figure to launch the movie player.

5.6 Extension to 3-D

The work described above simulates thermal flow across a two-dimensional region of space. Extending this simulation to a three-dimensional region is a logical extension of the work. Moreover, the mapping from the 2-D substrate to a 2-D reconfigurable device extends quite naturally to a three-dimensional reconfigurable substrate. Therefore, the basic 2-D heat equation element was re-designed as a three-dimensional element, intended to be tiled in three dimensions throughout a three-dimensional reconfigurable substrate.

The basic configuration of an element is the same as in the 2-D case:

- the current temperature of the element is stored internally in an 8-bit register;
- elements are arrayed in a three-dimensional layout, each element having 6 adjacent neighbors (North, South, East, West, Top and Bottom);
- elements copy their current temperature to their neighbors; and
- elements thus also receive temperature readings *from* each of their neighbors;

The behavior of each element is also similar to the 2-D case:

- the difference between each neighbor's temperature and the element's own temperature is calculated;
- the sum of those differences is computed;

- that sum is scaled by a constant multiplicative factor (this is the thermal coefficient for the material being modeled). This scaled sum represents the predicted *change* ΔT in the element's own temperature over the next (small) time interval; and

- after each element has computed its own ΔT, the elements update their internal temperature register by adding ΔT to it. The temperature itself is stored in a Master/Slave-style register, to prevent race conditions: the new register value is pre-loaded before the outputs are changed.

The resulting 3-D element is composed of $17 \times 17 \times 5$ cells, and is described in Figures 5.6 – 5.10. Each figure shows one of five planes that comprise the complete circuit. These elements are tiled side-by-side, i.e., a collection of $Z = 0$ planes are tiled in the $Z = 0$ plane of the reconfigurable device, same for $Z = 1$, etc.

Figure 5.6 shows the $Z = 0$ plane of the 3-D heat equation element. It is on this plane that inputs and outputs are exchanged to the north, south, west and east of the element. I/O to the element below this one also occurs on this level.

Temperature is stored as an 8-bit signed integer, and is passed from one element to another. However, partial sums and differences within the circuitry may exceed the range $[-128, +127]$. For this reason, internal buses and arithmetic blocks are *11* bits wide. Inputs are sign-extended from 8-bits to 11.

The $Z = 0$ plane's circuitry first adds the bottom element's temperature (labeled "B") to the eastern element's temperature ("E"). The western temperature ("W") is then added,

and this partial-sum is sent to the $Z = 1$ plane, which adds the top element's temperature "T" and returns this sum to the $Z = 0$ plane. This is then added to the norther element's temperature ("N") and, finally this is added to the southern element's temperature ("S"). This final sum is then sent to the $Z = 1$ plane.

The remaining circuitry on the $Z = 0$ plane is used for routing signals among the five layers of the complete circuit. For example, the element's temperature needs to be sent to each of the neighbors on the $Z = 0$ plane. This temperature is passed down from higher Z planes, along the perimeter of the circuit, until it reaches the $Z = 0$ edge outputs (e.g., the cells labeled "Southern Input/Output").

The $Z = 1$ plane is shown in Figure 5.7. This plane responsible for the following calculations:

- receive shifted versions of the element's temperature ("Temp") from the $Z = 2$ plane — specifically, $2 \times Temp$ and $4 \times Temp$;
- compute $6 \times Temp$ by summing $2 \times Temp$ and $4 \times Temp$;
- compute this element's change in temperature $DT = B+E+W+T+N+S-6\times Temp$; and
- transfer DT to the $Z = 2$ plane for further processing.

The $Z = 1$ plane also performs part of the calculation of $B+E+W+T+N+S$: specifically, it adds T to $B + E + W$.

The plane at $Z = 2$ performs the following actions:

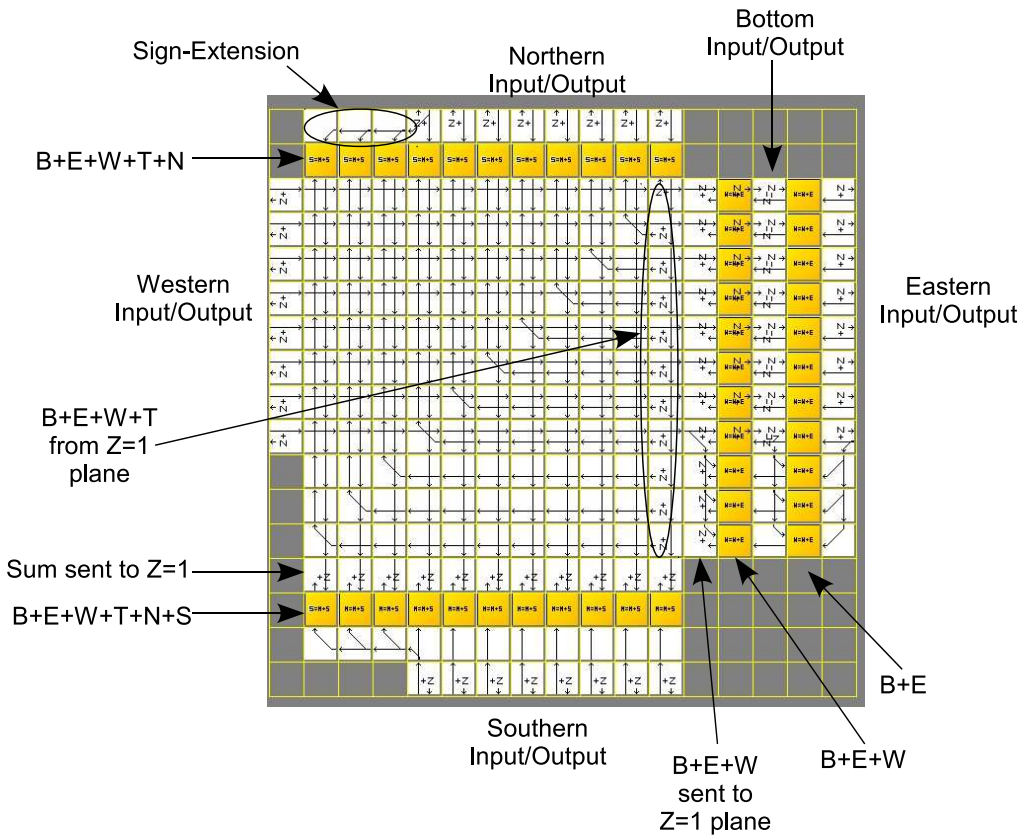

Figure 5.6: Z=0 Plane of 3-D Heat Equation Element. Temperature readings from the north, south, west, east, top and bottom elements are designated "N," "S," "W," "E," "T" and "B" respectively. I/O with neighbors uses 8-bit integers, which are sign-extended to 11 bits internally (to prevent overflow). This plane mainly calculates the sum of the temperature from all neighbors (part of the calculation is performed on the $Z = 1$ plane). The final sum is sent to the $Z = 1$ plane for further processing.

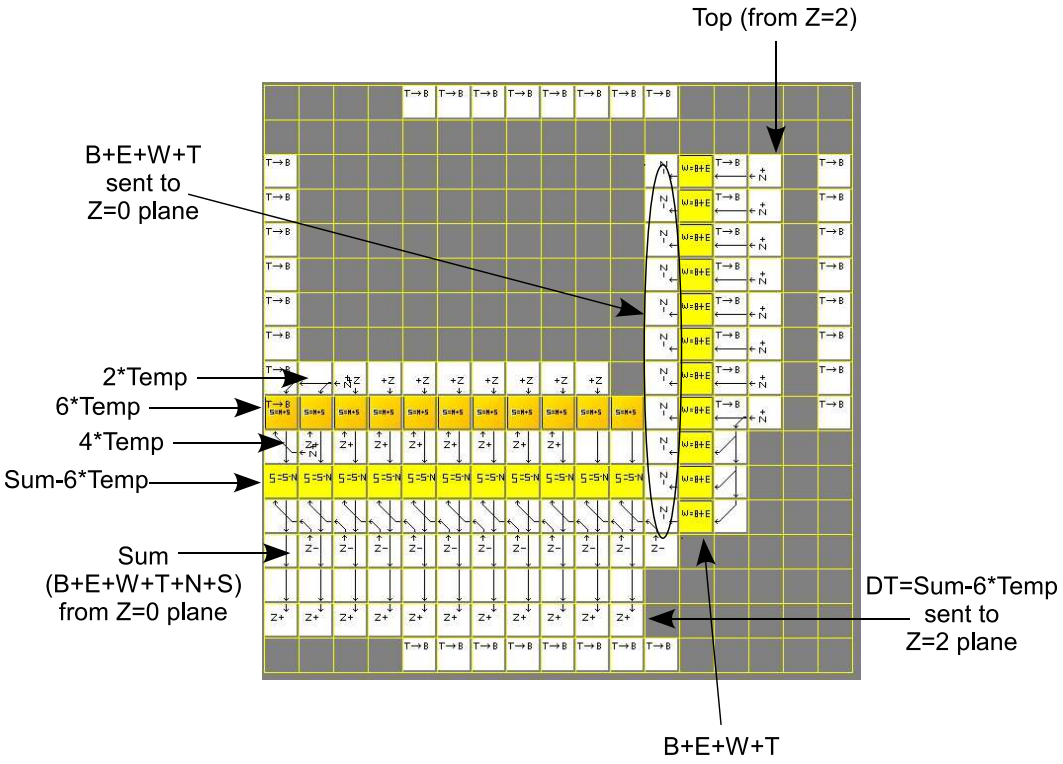

Figure 5.7: Z=1 Plane of 3-D Heat Equation Element. This plane subtracts $6 \times Temp$ from $B+E+W+T+N+S$ to compute the element's change in temperature DT. DT is passed to the $Z = 2$ plane for further processing.

- receive the element's temperature("Temp") from the $Z = 3$ plane;
- shift Temp left one bit, and send to the $Z = 1$ plane;
- shift Temp left two bits, and send to the $Z = 1$ plane as well;
- receive DT from the $z = 1$ plane; and
- compute the element's new temperature as $DT + Temp$ and send to the $Z = 3$ plane.

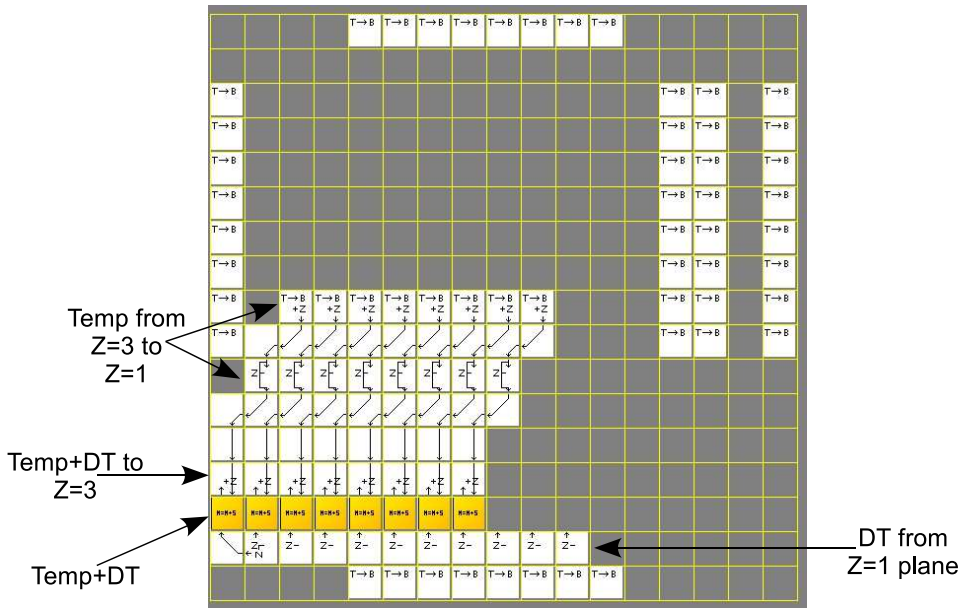

Figure 5.8: Z=2 Plane of 3-D Heat Equation Element. This plane helps compute $6 \times Temp$, and also computes the element's new temperature=$Temp + DT$.

The $Z = 3$ plane contains the register that stores the element's temperature. It also contains a *pulse generator*, comprised of a single cell (called a "crystal") whose truth table contains a pattern of alternating 1s and 0s; along with a cell (called a "crystal controller") that continually reads the crystal's truth table and sends the sampled bits to its output. This thus produces a stream of 1s and 0s, which are fed to the CLOCK input of the Temperature Register. On the rising edge of the clock, the element's new temperature ($Temp + DT$) is pre-loaded into the register; on the falling edge of the clock, the register's outputs reflect the new value. The temperature is also distributed to the planes above and below this one.

The $Z = 4$ plane is the top layer of the heat equation element. This is the layer that exchanges temperature information with the element above itself ("Top Input/Output"). it also receives the element's own temperature from the plane below it, and fans it out to the four edges of this plane. At each of those edges, the temperature is sent to the $Z = 3$ plane; each plane passes it to the plane below it, until the $Z = 0$ plane, that outputs the temperatures to neighboring elements on the same $Z = 0$ plane.

5.7 Comment on 3-D wiring efficiency

It should be noted that even though the 3-D heat equation element works with six inputs; requires wider adders and registers; and has to perform more-complicated arithmetic (multiplication by 6 vs 4), this element uses roughly 35% the number of cells as the simpler 2-D element. This decrease in cell count is directly attributable to the improved wiring density

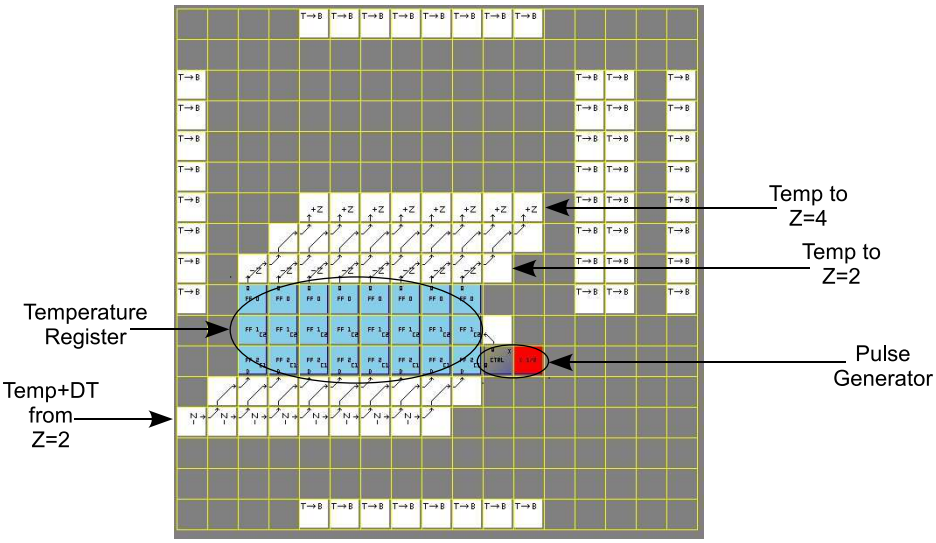

Figure 5.9: Z=3 Plane of 3-D Heat Equation Element. This plane contains the register that stores the element's temperature, as well as the update circuitry for loading the element's new temperature at each time step. The Pulse Generator generates a pattern of 1s and 0s that are fed to the register's clock input. Each uptick/downtick pair of pulses on the clock input causes the register to be updated with the element's new temperature.

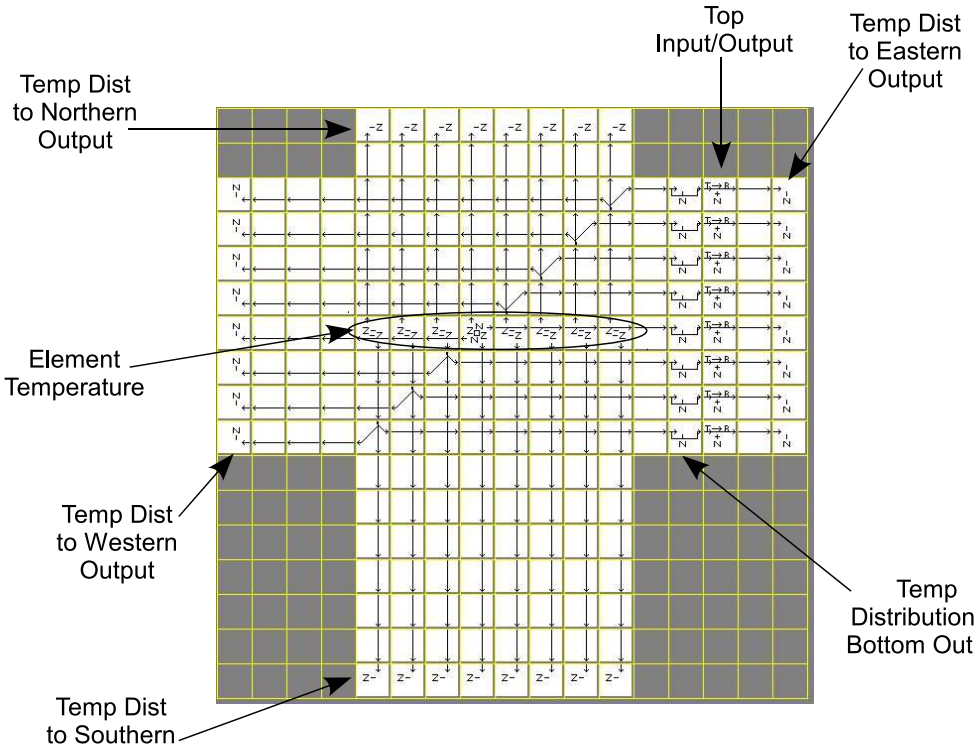

Figure 5.10: Z=4 Plane of 3-D Heat Equation Element. This plane communicates with the 5-plane element directly above it. It also distributes the element's own temperature to the edges of the plane, where it is passed down to lower planes, until it reaches the $Z = 0$ plane, that sends the temperature to its neighbors.

afforded by building circuits on more than one plane. The improvement is similar to having, for example, two metal layers inside a silicon chip as opposed to only one. The ability to "step over" wires instead of routing around them can greatly increase the resulting density of circuits implemented on a 3-D platform.

This also raises another important observation. Architecturally, a reconfigurable system such as we've been working with here can be built as a "true 3-D" device, i.e., one with comparable dimensions in all three axis. Manufacture of such a 3-D substrate is problematic though, not only in terms of manufacturing, but in terms of issues such as intra-circuit noise and excessive heat buildup [**?**]. However, as this example has shown, there can be great benefit to a *this* 3-D substrate, i.e., one composed of only a few (very large) 2-D sheets. In other words, a 3-D substrate whose x- and y-dimensions are potentially quite large, but whose z-dimension is very small (but larger than 1).

One of the impediments to 3-D manufacture is creating connections from one layer to another, e.g., inter-layer vias [**?**]. A fully-connected architecture containing trillions of elements per sheet would require trillions of inter-layer vias between each sheet — something that is currently beyond our manufacturing capabilities. **However**, it is possible to manufacture a network of cells *on a 2-D substrate, but interconnected in a 3-D topology.* Figure 5.11 shows how to implement a three-layer 3-D topology on a 2-D substrate. The wiring is straightforward, with the tops and bottoms of elements interconnected as needed to create a 3-D topology.

This figure shows a single $1 \times 1 \times 3$ element, intended to be tiled in the $X - Y$ plane — and

in so doing, the dimensions of the overall circuit will grow in X and Y, but will remain fixed in Z (a height of 3 layers). The tiling is done such that, for example, one element's $E_{Z=0}$ output is connected to a neighboring element's $W_{Z=0}$ input, and so on. Figure 5.12 shows a 4×4 tiling of these elements, resulting in a 3-D array whose dimensions are $4 \times 4 \times 3$.

This will not work for a general 3-D system, since the layout shown in Figure 5.11 is not scalable in Z: to extend that layout from three layers to four requires a complete re-mapping of the 3-D layout onto the 2-D substrate. However, the layout **is** scalable in two dimensions (X and Y). So in the case where one is satisfied with a fixed number of planes and only wishes to scale in the X and Y axis, the (thin) 3-D substrate can be manufactured using conventional 2-D manufacturing, while maintaining 2-D scalability (meaning two substrates can be connected on their edges to create a larger substrate, without re-wiring anything inside the substrates). Thus, manufacturing a 3-D substrate which is "three-dimensional-enough" to make routing easier and more-dense is already within current capabilities.

The next chapter will use this heat equation solver to compare the architecture proposed in this paper with contemporary architectures, based on a number of issues including configuration time, sensitivity to errors, and run-time performance.

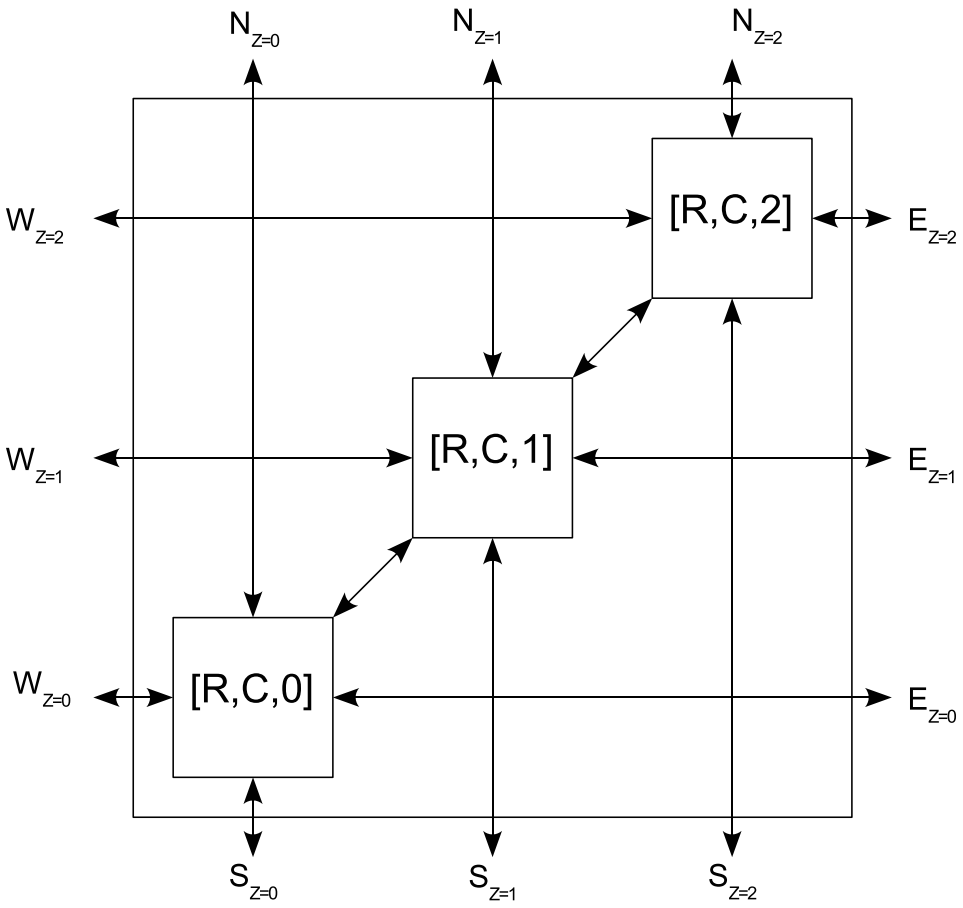

Figure 5.11: A single $1 \times 1 \times 3$ element that can be scaled in the $X - Y$ plane to produce an arbitrarily-large $X \times Y \times 3$ 3-D array. Note that all three planes are **physically co-planar**, but are wired together as independent planes.

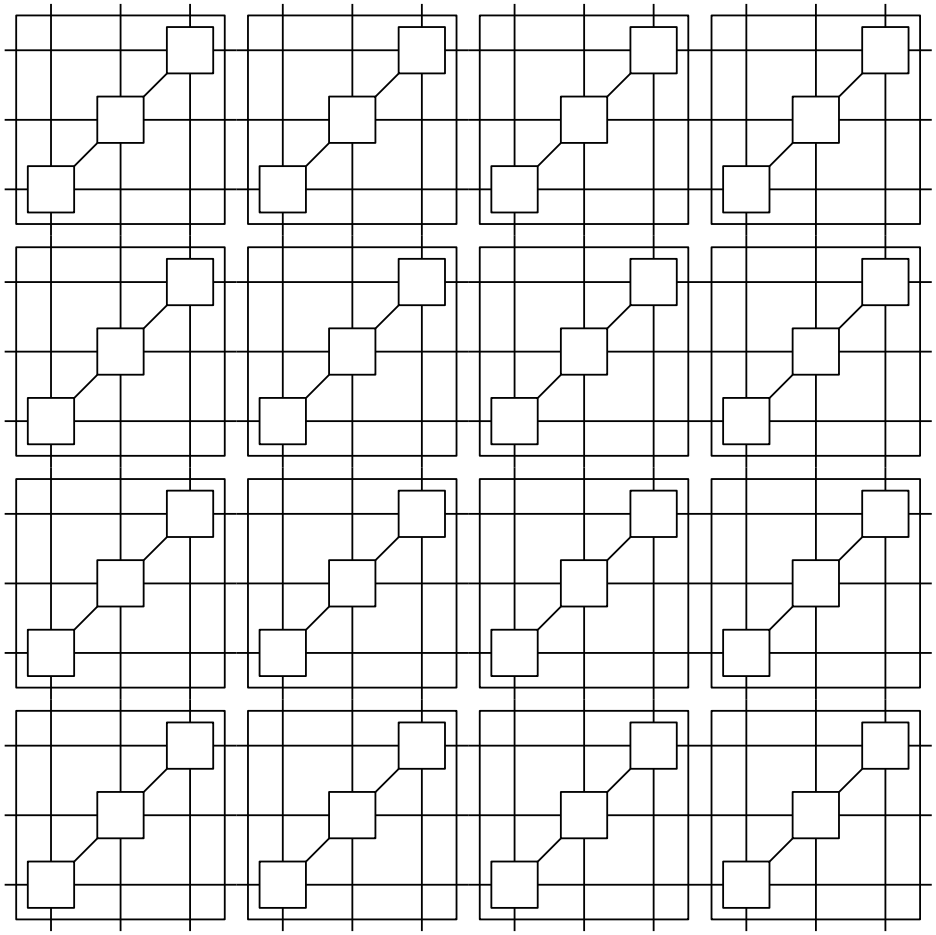

Figure 5.12: $4 \times 4 \times 3$ array of cells, comprised of a regular tiling of a scalable $1 \times 1 \times 3$ sub-array. Note that if four of *these* arrays are tiled in X and Y, the result will be a large $8 \times 8 \times 3$ array. However, there is no way to connect these elements so as to produce a circuit containing more than three planes. To do so, internal connections much be rewired.

Chapter 6

Comparative Analysis

Conducting a meaningful comparative analysis of the proposed self-configurable architecture with contemporary computing architectures is challenging. A generally-useful system based on the proposed architecture is not presently viable with today's manufacturing technologies. This is primarily because of the scale of the system being proposed. Nonetheless, there are good reasons, as explained below, for working towards a very large-scale self-configurable system.

6.1 How Large?

Much of this work draws inspiration from nature [**?**]. In this area, it is informative to consider what sorts of *atomic units* are used to build larger-scale systems. In biological systems, the basic building block is a *cell*. A single organisms may contain hundreds of trillions of cells,

each of which is itself a complex mechanism containing subsystems for functions such as replication, repair, power generation and molecular synthesis [**?**]. It is collections of these complex, subsystem-rich building blocks that comprise multi-cellular organisms.

Compare this to human-engineered systems such as single-chip CPUs or complex computing engines. In such systems, the elemental building block is basically the transistor — an effectively single-function device (to act as an electronically-controlled switch). FPGAs utilize a slightly-larger building block (e.g., CLBs in a Xilinx FPGA [**?**]), capable of storing a small amount of information, performing a programmed input-to-output mapping, and perhaps taping into more-complex blocks such as floating point units. These represent a relatively simple building block compared to what nature has developed.

A more-complex building block for human-engineered systems could contain many more sub-systems, such as those discussed in this work, including sub-systems for:

- checking hardware behavior and detecting faults;
- isolating defective regions of the hardware;
- configuring new circuits within the substrate;
- moving sets of circuits from one region of the substrate to another;
- finding pathways among separate circuit; and
- monitoring circuit used and triggering hardware reassignment as needed.

One Element	10^4 transistors
Single Cell	10^6 elements=10^{10} Transistors
Complete Organism	10^{14} cells=10^{24} Transistors

Table 6.1: Component Counts for a Theoretical System/Organism

One way to think of these building blocks is as a *smart transistor*: a device with a simple basic function (e.g., switching), but endowed with a lot of additional capability.

Following the example of biology, a much larger atomic unit could be constructed, encompassing these and other features. This has been done on a much smaller scale, in the development of *embryonic stem cells* [**?**, **?**]. In this preliminary work, approximately 73,000 simple reconfigurable elements were used to build a single "supercell." A single three-dimensional reconfigurable element can be composed of roughly (to the nearest order of magnitude) 10,000 transistors. A typical human contains 10^{14} cells; assuming each cell is comprised of 10^5 reconfigurable elements, and each element is comprised of 10,000 transistors, this amounts to 10^{23} transistors. If we allow for a cell that is roughly $10x$ more complex than the one described above, this brings us to an estimate of 10^{24} transistors for a large "organism" comprised of feature-rich building blocks. This is a useful target for the proposed architecture. Table 6.1 summarizes this size data.

A different approach is taken by [**?**], in the creation of a large neural network comprised of one billion neurons. While these goals of that work differ from that of the architecture presented herein, the idea of using a large number of very coarse-grained building blocks is similar, as is the bio-inspiration that is behind the work.

6.2 Viability

An Avogadro-scale system — one containing 10^{24} transistors (or perhaps 10^{24} *reconfigurable elements* or some other atomic-unit) — represents an increase of approximately 15 orders of magnitude in transistor count. According to Moore's law, this won't be achieved for at least another 75 years ($1.5 \times log_2(10^{15})$). However, manufacturing a fine-grained self-configurable system at an Avogadro scale may well be feasible significantly sooner, but only along a different development path. The reason is due to a shift in architecturally-based manufacturing requirements.

The historical advancements in chip manufacturing have been in several areas, most notably:

- transistor size — the measure by which fabrication technologies are named, i.e., a 45μ process;
- switching speed, which is beginning to level out, but has increased significantly over the past two decades [**?**];
- power requirements per transistor, which has necessarily decreased over time in order to keep total system power consumption at a manageable level; and
- yield improvement, allowing cost-effective manufacturing of larger die containing more transistors from generation to generation.

These are some of the key factors that drive progress in contemporary fabrication, and have kept the semiconductor industry's progress in line with Moore's Law. However, the

importance of these factors is different for the architecture proposed herein:

- transistor size — this is certainly still an important factor, but by itself is insufficient;
- switching speed — rather than achieving system speed by increasing the speed of each transistor, system speed is obtained by the parallel operation of a huge number of sub-circuits. Each sub-circuit may be relatively slow, but system speed is dominated by the *number* of such sub-circuits. This is certainly a bio-inspired idea: for example, in the human brain, neural operations run at approximately 100 – 1000 Hz [**?**];
- power requirements per transistor — if the switching speed of each transistor is reduced, the power requirement will also be reduced, by a similar factor. This is another idea that mimics what is found in nature. The human brain contains 100 billion neurons, yet consumes only 20 watts [**?**] — less than half the power used by a Pentium 4 [**?**]; and
- yield improvement — as described above, perfect fabrication is not necessary in a locally-controlled, nearest-neighbor array of simple homogeneous elements.

For this architecture, there is one central requirement: **the ability to manufacture a large number of elements** (each comprised of a simple combinatorial circuit-plus-memory) **in an extremely dense package.** So while transistor size is important, in some sense, what is most important is usable transistor density: how many meaningfully-interconnected switches can be put into one package or system. And because this is a critical factor, almost certainly

the best manufacturing will be one that produces a three-dimensional chip. Assuming a $1nm \times 1nm$ transistor size, and ignoring interconnect, placing 10^{24} transistors on a square 2-D substrate would require 10^{12} transistors per edge. That means a die size of **1 square kilometer**. In contrast, assuming a transistor size of $1nm \times 1nm \times 1nm$, those same 10^{24} transistors could be arranged in a cube with 10^8 transistors per size, resulting in a die size of only 1000 cubic centimeters (one liter).

While manufacturing a liter-sized circuit with today's fabrication techniques is unthinkable, such targets may be feasible in the future, with technologies such as quantum-dot computing; single-electron transistors; holographic computation; atomic-scale fabrication; or crystal growth-based fabrication [**?**, **?**, **?**, **?**, **?**].

One may question the basis for exploring fabrication that is so radically different from today's standards. And while a dense, imperfect 3-D system of slow switches may seem an unlikely build target, there is at least one precedent: nature. As an example, the human brain is just over a liter in volume; contains roughly 10^{11} neurons (which are more complex than a single transistor); 10^{15} connections among those neurons; operates with a modest neuron clock rate of 200Hz; and is certainly not defect free (brain cells die from a variety of causes).

The question here is thus as follows: assuming a radical breakthrough in manufacturing capability — one that, perhaps, more-closely resembles natural systems — what are the merits of this proposed architecture relative to other existing architectures? With this in mind, the following analysis considers the benefits of four different architectures, assuming such a manufacturing capability. The four architectures are:

- a massive multicore system;
- a massive cluster of uniprocessors;
- a coarse-grained, hierarchically-routed, externally-configurable FPGA; and
- a fine-grained, regularly-interconnected, self-configurable array.

6.3 Speed

For a comparative analysis of speed across different architectures, it's useful to consider the simulation of heat flow based on a finite element method, as described previously. This assumes a square substrate, divided into a uniform 2-D mesh of elements, arranged as an $n \times n$ array. Under these assumptions, one can estimate the speed of a single simulation step, i.e., the time required to update the state of each element in the mesh, to determine its temperature at the next time step.

For a scalar processor (CPU/memory model), the next state of each node will need to be computed one at a time. The basic update loop is a relatively simple integer computation: 5 additions/subtractions; one multiplication ($\times 4$); one division ($1/8$); four fetches from memory; and one write to memory. Assuming for simplicity that all data is stored in physical memory (no virtual memory); a theoretical cycle time of 1 nSec (with no memory wait cycles required); and ignoring overhead for index updates and so on, this gives a total time of 12 nSec per element, for a total of $12n^2$ nSec to update the state of the mesh for a single timestep. For a trillion (10^{12}) elements ($1,000,000 \times 1,000,000$), this is 12,000 seconds (3.33

hours) per timestep.

A multi-core CPU can do better than this, since each core could be used to update an element, allowing multiple simultaneous updates. Assuming m cores are available, this provides a maximum speedup of $m\times$ over a scalar processor. However, unless massively-multi-ported memory is available, a multi-core CPU will have issues of memory contention. Each element needs to read state information from four neighboring elements (in a 2-D arrangement). This relatively-limited amount of required interconnect may be helpful, but does not solve the issue that n^2 *threads will be trying to access memory simultaneously in order to perform a state update on the element they are assigned to.* This effectively reduces the multi-threaded processing to a scalar one, where element states are updated one at a time. Some pipelining can be achieved — for example, one element can be summing its neighbors temperatures while another element is accessing memory to read its neighbors temperatures. But even with an ideal pipeline that allows 12 simultaneous threads without any memory stalls, the effective speed of computation is one update per time unit (1 nSec), and the total update time for the entire mesh would still be 10^{12} nSec, or 1,000 seconds (around 16 minutes).

A cluster of CPUs, each with its own memory, improves the situation more, since each CPU can access its own state information simultaneously with all other CPUs (assuming one CPU per mesh element). But there is still a potential resource contention as each CPU tries to read its neighboring elements' temperatures. Unless the CPUs have been interconnected in the same nearest-neighbor topology as the finite element mesh's elements, they are probably sharing some sort of common communication channel. This may offer better concurrent

access than a single shared memory, especially with pre-fetching or local caching/update of neighbors' memory; but the same basic problem persists: the larger the number of elements, the longer the mesh update process will require. For a communication bus that completes only one data transfer at a time, the total mesh update time would be on par with the multi-core case (1,000 seconds). Increases in the simultaneous-data carrying capacity of the bus would reduce that time by roughly the degree of parallel transport available on the bus.

The above issues can be viewed as a two-fold inefficiency:

- there is no dedicated communication between neighboring processing units: this leads to resource contention; and
- the processing units themselves are far more powerful than they need to be.

These two factors in fact stem from the same underlying problem: a sub-optimal use of the transistor budget. Were it possible to re-allocate most of the transistors that constitute an entire CPU, and use them instead to create direct communication paths between neighboring processing units (each such unit being a very simple state machine), one might achieve a significantly faster simulation.

A reconfigurable array fits this solution nicely (depending on the available interconnection strategy between configurable elements). Assuming complete connectivity can be established among each set of neighboring processing units, and still assuming 12 nSec update time for a single element, the time to update the state of the entire array *is still 12 nSec*, because there is no memory- or bus-contention that would cause an element's update to stall. For

Scalar CPU	3.33 Hours
Multicore	16 Minutes
Cluster	16 Minutes
Reconfigurable	12 nSec

Table 6.2: Runtime for heat equation simulation, single timestep: 10^{12} elements; 1 GHz clock

a trillion-element mesh, this represents a speedup of roughly 11 orders of magnitude over a multi-core or multi-CPU solution.

Of course, to achieve full speedup will require complete connectivity among each set of neighboring elements. If this is not the case, the simulation time will be increased by roughly the degree to which connectivity resources need to be shared. For this reason, a fine-grained reconfigurable device whose internal communication paths are also fine-grained would be a very desirable architecture, vs. a more coarse-grained device with more-limited routing resources. Table 6.2 summarizes these results.

6.4 Reliability

In discussing reliability of future systems, it should be noted that fault rates depend on many things besides simply transistor count: transistor *size*; operating voltage; and fabrication technology each have an effect on fault rates. Reliability is also a function of system design, both at a hardware and a software level.

It's also important to discriminate between two general classes of errors: *manufacturing defects*, that occur when a device is built, and are effectively permanent; and *upsets*, that

occur during the operation of a device, and take the form of a bit value that is opposite of its correct value. Furthermore, upsets may be temporary, appearing as a momentary glitch in a signal; or they may be semi-permanent, such as when they occur in a register or other memory location. Semi-permanent upsets can be cleared by re-initializing the system (power-cycling, etc.) or by re-loading the correct bit value; while defects are generally considered permanent.

6.4.1 Manufacturing Defects

Accurately predicting yield rates for an as-yet-unknown fabrication process, particularly one envisioned as "post-singularity," is unlikely to result in more than a best-guess. It's clear, however, that *yields rates generally drop as transistor count and density increase* [**?**]. Historically, VLSI fabrication has relied on achieving a defect rate near zero [**?**]. But for a system with the ability to self-analyze and self-configure, there is an alternate to perfect fabrication: one can *adapt to doing engineering on defective devices*, i.e., lay out designs on a substrate despite the *expectation* of defects in that layout. In other words, changing the question from "how will manufacturing defects be eliminated?" to "how will circuits be operated successfully in the presence of defects?"

For conventional CPU/memory systems — multi-core processors or clusters of uniprocessors — there is no clear answer, as current manufacturing relies on perfect fabrication. Defects in on-chip cache might be manageable via ECC or other error-correcting codings, but defects in other circuitry can generally not be managed, as there's no way to dynamically adjust

a VLSI layout to avoid defective regions of the substrate. Constructing 10^{24} CPUs (or cores) will thus likely require manufacturing of separate subsets of CPUs, with subsequent interconnection of them: a massive single-die system composed of 10^{24} cores or CPUs is simply not feasible without an *extremely* pure manufacturing process.

Conventional FPGAs might fare better. A manufactured array can be pre-tested for defects; a defect map can be supplied to the development tools; and the layout of a circuit on the FPGA can be done so as to avoid defective blocks. This is similar to how the Teramac project dealt with defects [**?**]. One potential drawback is the relatively large number of critical points within an FPGA's design: a defect in clocking lines, communication lines, or the configuration subsystem may render the entire device unusable.

In a locally-connected self-configurable architecture, there are very few critical points. If a system-wide clock is distributed from element to element to element, a defect on that distribution bus could impair the entire device. However, a locally-generated clock can be used, with the clock generated inside each element or each block of, say, 1000 elements. Alternative clocking schemes, such as using a laser pulse to optically trigger cell configuration, would obviate concern over, say, disabling the entire system by shorting a clock line. Power lines are another global signal, and thus represent a potential critical failure point. Again, non-conventional power distribution could be used, such as distributed magnetic pumps driven by a fluctuating electromagnetic field surrounding the entire device — similar to something proposed for powering a *paintable compyter* [**?**]. While these embellishments could theoretically be applied to conventional FPGA designs, the relative irregularity of

those designs poses an additional challenge.

Moreover, as shown previously, local configuration can be used to perform efficient defect testing *at configuration time*: an initial set of known-good elements are used to test nearby elements; newly-discovered good elements are incorporated into the testing circuit, allowing more elements to be tested in parallel; and as more elements are discovered to be good, those too are added to the test circuit, creating an even larger test circuit. In this way, the test speed actually increases as the test continues. For a 3-D array, 10^{24} elements can be tested in roughly 3×10^8 timesteps [**?**].

6.4.2 Runtime Upsets

While contemporary reconfigurable devices tend to use SRAM for storing lookup tables [**?**], DRAM tends to be denser, and thus may be more prevalent in future large-scale reconfigurable systems. Typical DRAM upset rates are on the order of $10^{-10} - 10^{-17}$ errors/bit-hour [**?**]. Optimistically assuming the lower rate, and a modest memory size of 128 bits per node (that's a typical configuration memory size for an FPGA, and a conservatively-small memory size for a CPU/memory system), this amounts to 128×10^{24} bits, giving a total of 1.28×10^9 errors per hour, or a MTBF of 2.8 μsec. Memory upsets will occur, *somewhere in the system*, very frequently. If such upsets occur independently and are well-distributed throughout the system, then it should be possible to retain, say, three copies of each memory, and, in the event of an upset, use a majority rule to determine the correct value. This value can then be written back to correct the upset. However, given the large number of memory locations

to be checked (over 10^{26}, and the speed with which errors must be detected and corrected (2.8 μsec or less), it is unlikely this detection and correction process can be managed from any one centralized location. Instead, it will need to be distributed throughout the system, with multiple checks and corrections occurring simultaneously. Of the architectures we are considering, only the locally-controlled, self-configurable one is suited to this task. In fact, it has been shown that a simple circuit can continually monitor configuration information, detect upsets, and replace them on-the-fly, without disrupting the run-time operation of the circuit being scrubbed [**?**]. This ability to tolerate run-time defects is increasingly important, as soft error rates tend to increase as transistor size decreases [**?**].

6.5 Setup Time

There are applications — cellular automata, finite element analysis, search problems, simulation of physical systems — that can be efficiently solved by naturally partitioning the problem space into a number of identical regions, and mapping those regions to a system of identical processors. This is one way to take advantage of massive parallelism within a system. In such a case, where the large-scale system will be comprised of a number of identical sub-systems, it may be possible to bootstrap the system more-efficiently by configuring all sub-systems in parallel with one another. Doing so requires some sort of partitioning of the system into blocks that will be configured identically.

Massive multicore systems or clusters of CPU/memory systems have a natural partitioning, and thus a natural vector for parallelization of their bootstrapping. Assuming each proces-

sor's memory is to be configured identically (with possible subsequent differentiation via e.g. positional information), the setup time for these systems is on par with the setup time for today's single-node systems — typically a minute or two to bootstrap from power-on, and a few seconds or less to load a particular program into memory.

For FPGAs, the startup time varies roughly with the length of the configuration string. But unlike the above case, there is no canonical partitioning of the device's elements. If, for example, one is configuring a circuit comprised of one million identical sub-circuits, and each sub-circuit requires only 1,000 bytes of configuration information, those same 1,000 bytes will be loaded into the device one million times. In this particular example, this is very inefficient.

Were the FPGA partitioned into blocks, with each block being configured identically, then a shorter configuration string could be used to bootstrap the entire device. But how should the device be partitioned? One could choose, at manufacture time, a particular partitioning of the reconfigurable elements. For example, reconfigurable elements could be organized in groups of 1024×1024 elements, with all groups being configured identically to each other, in parallel. If there were a million such groups, one would only need to supply the relatively-short configuration string necessary to configure a single group; this string would be distributed to all groups, and they would be configured identically, in parallel with each other. This would result in a much more efficient configuration than using a configuration string that independently specifies the configuration of each reconfigurable element.

Such a manufacture-time decision will, in some cases, result in wasted resources (for example,

CPU-Based Systems	2 Minutes
FPGA	278 Hours
2-D, Self-Configurable	50 Minutes
3-D, Self-Configurable	300 mSec

Table 6.3: Startup/Configuration time for system with 10^{15} elements: 1 Ghz clock

if the circuit being run in parallel requires only a 16×16 element block — enough for a simple pattern-matching circuit); and in other cases will be insufficient for very complex circuits requiring more elements than contained in one block. There is no single way to partition the reconfigurable elements that will be optimally suited to all problems. And in the absence of any such partitioning, the configuration time will grow according the the size of the configuration string, which grows according to the number of reconfigurable elements.

With a self-configurable system though, the partitioning can be decided not at manufacture time, *but at run-time*, based on the problem that is to be solved. And, as shown previously, the configuration of a bootstrap system based on the desired partitioning can also be performed efficiently — again, on the order of 3×10^8 timesteps for a 3-D device containing 10^{24} reconfigurable elements; while a 2-D device would require on the order of 3×10^{12} timesteps. This architecture thus offers a great deal of flexibility in the kinds of circuits that can be efficiently implemented on it. These results are summarized in Table 6.3.

6.6 Cross-Sectional Bandwidth

For multi-core systems or a cluster of CPUs, the cross-sectional bandwidth is highly dependent on the interconnection topology among the CPUs or cores. For a ring topology,

for example, the bandwidth will be limited to the bandwidth of a single link; in particular, it does not automatically scale as the number of nodes increases. For a bus network, the bandwidth may be higher, depending on the width of the bus, but again does not, in general, scale-up with an increase in nodes/cores. A fully-connected mesh may allow a scaling of the bandwidth with the number of nodes, but only if the cores have sufficient address space to specify an increasing number of destinations. Thus, scaling up the bandwidth may involve a re-architecting of the individual nodes or cores.

In an FPGA, the routing structures are closely tied with the configurable blocks (CLBs), and thus tend to scale up as the number of CLBs increases [**?**]. In particular, the General Purpose Interconnects (GPIs) allow adjacent blocks to communicate directly with one another. Thus (theoretically), for 10^{24} CLBs, arranged in a two-dimensional array of $10^{12} \times 10^{12}$, a maximum of 10^{12} words can be transferred in parallel, for a bandwidth on the order of 10^{12} (the actual bandwidth depends, of course, on the speed with which bits can be broadcast across whatever physical interconnect is used).

In a three-dimensional (self-configurable) array of elements connected with a 6-neighbor topology, 10^{24} elements would be arranged as a $10^8 \times 10^8 \times 10^8$ collection. A single slice through this cube of elements would expose faces comprised of 10^{16} elements each (arranged as $10^8 \times 10^8$ elements), with each element directly connected to an adjacent element on the other side of the slice. Thus, by virtue of being three-dimensional as well as fully-locally-interconnected, the cross-sectional bandwidth of this architecture would be on the order of 10^{16}. Table 6.4 summarizes these results.

Scalar	1
Multicore	min(# Cores, Network BW)
Cluster	min(# Nodes, Network BW)
FPGA	10^{12}
3-D Self-Configurable	10^{16}

Table 6.4: Order of Cross-Sectional Bandwidth

6.7 Synopsis

Table 6.5 collects the analysis from this chapter. As can be seen, other than for setup time, reconfigurable arrays perform at least as well as scalar machines, clusters of CPUs, and multi-core systems. Among reconfigurable arrays, a locally-connected self-configurable array is better-suited to handling defects and errors, and offers a faster setup time, than a contemporary FPGA. Cross-sectional bandwidth may also be improved, but primarily because a locally-connected self-configurable array easily scales as a 3-D system, whereas most FPGAs are still primarily two-dimensional.

	Scalar Machine	Cluster	Multi-core	FPGA	Locally-Connected Self-Config Array
Speed (time for one HE sim step, 10^{12} elements)	12,000 sec	1,000 sec	1,000 sec	12 nsec	12 nsec
Manufacturing Defects	Extremely likely and critical	Extremely likely and critical	Extremely likely and critical	Some avoidable, some critical	Mostly avoidable, few critical
Run-time Upsets	Short MTBF	Short MTBF	Short MTBF	Short MTBF	Run-time scrubbing
Setup Time	Short, indep of size	Short, indep of size	Short, indep of size	$O(n)$ for n CLBs	$O(n^{1/3})$ for n elements (3-D array)
Cross-Sectional bandwidth (10^{24} nodes)	—	network-dependent	network-dependent	up to 10^{12} (2-D array) but possible address space issues	10^{16} (3-D array)

Table 6.5: Architecture Comparison

Chapter 7

Conclusions and Future Work

7.1 Conclusions

A computing architecture has been proposed that possesses the following four characteristics:

1. it is a reconfigurable system — the behavior of its fixed hardware can be modified via a programming/configuration operation;
2. the system is able to introspect — examine its own state and configuration — and respond by modifying its own state and configuration;
3. the parts of the system that implement internal operations such as bootstrapping are themselves composed of reconfigurable elements; and
4. there is no inherent hierarchy among elements that *configure* others and elements that are *configured by* others.

These characteristics have been examined in detail, and the specifics of an architecture that embodies them has been presented. This architecture has been shown to be particularly

useful in systems comprising a very large number of reconfigurable elements. The system has also been show to be feasible for managing the complexity of such extremely large systems.

Examples have been given to illustrate the unique nature of this system, including the use of internally-directed configuration to efficiently customize a bootstrap system on-the-fly. This bootstrap system has been extended to test regions of reconfigurable elements for defects, and avoid defective regions, this allowing the use of imperfect substrates. Moreover, the ability to test multiple regions in parallel has been demonstrated.

The use of a cascaded bootstrap has also been shown: first a one-dimensional line of blocks is configured; these are then used to configure a two-dimensional region, with all elements in each new one-dimensional line being configured in parallel; and finally, the finished two-dimensional region is used to configure a three-dimensional region, with all elements in each new two-dimensional region being configured in parallel.

Chapter 4 presented specific structures and circuits for generating bit patterns such as those used to perform the above configurations. These structures and circuits are part of a general methodology for generating bitstreams, including repeated generation based on counters, and conditional-generation based on the value of internal signals. This demonstrates the feasibility of implementing entirely-self-contained systems that require minimal external intervention.

The utility of a large system of reconfigurable elements to solve practical problems has been examined, using the heat equation as a sample problem. Circuits have been presented for simulating heat flow through a three-dimensional region of space by:

- mapping regions of physical space to regions of the reconfigurable system;
- configuring blocks within the reconfigurable system to simulate heat flow through a small homogeneous region;
- setting initial and boundary conditions within the simulated medium; and
- stepping the simulation multiple times, to simulate the flow of heat among the elements of the material being modeled.

This approach successfully models heat flow as expected.

While full-scale three-dimensional fabrication is presently an elusive manufacturing goal, the usefulness of a "thin three-dimensional system" — i.e., one with arbitrarily-large extend in the X and Y axis, but only a few layers in the Z direction — has been shown, specifically for simplifying the routing of essentially two-dimensional circuits. Moreover, a design has been presented for a thin three-dimensional substrate that preserves perfect architectural scalability in the X and Y directions. Combined with the architecture's inherent fault isolation, and the ability to test-for and manage defects, this suggests that manufacture of a large-scale version of the proposed system may be feasible.

Finally, a comparison has been made — with respect to the setup and execution of the sample application — on the performance of the proposed system relative to a scalar CPU/memory system; a multi-core CPU/memory system; a cluster of CPU/memory systems; and a more-contemporary (two-dimensional, not self-configurable) reconfigurable device. Of these architectures, the proposed system is the only one that performs well in *all* the categories considered.

7.2 Future Work

Most work on the proposed architecture is presently done with simulators. This presents a fundamental challenge: the architecture is best-suited to implementing circuitry that has a high degree of parallelism; but these are precisely the cases that are least efficient to simulate on a scalar processor. Some work was done on distributing the simulator over a small number of nodes, in the hope of reducing the overall simulation time. Unfortunately, inter-simulator communication dominated the total execution time, and no speed-up was observed. However, because connectivity among reconfigurable elements is strictly between nearest neighbors, the inter-simulator overhead should be fairly constant regardless of the number of nodes (assuming the interconnect-network has sufficient bandwidth). Therefore, for a sufficiently-large number of nodes, the parallelism of the simulation should begin to reduce simulation time more than the overhead increases it, and the multi-node simulation should be faster than a scalar one. Exploring this more-deeply, determining the number of nodes necessary for an overall speed-up, and confirming this experimentally, would be

interesting and useful.

Fabrication of the proposed system remains the biggest hurdle, and looking at emerging technologies to find one suitable for the specifics of the proposed system is an important task. Note that fabrication itself is not the difficulty; the challenge is in fabricating a system with a sufficient number of reconfigurable elements to make the system useful. Since elements are used for simple functions such as logic or basic data transfer, a very large number of reconfigurable elements are required for implementing a large circuit. For static circuits, i.e., circuits that do not self-configure, there is no incentive to pay this huge overhead; but circuits that employ self-configurability tend to be large. In other words, the proposed architecture becomes most viable only beyond a certain number of elements — possibly one trillion. This is a formidable manufacturing goal. It's also difficult to "creep up to": there is little payoff in manufacturing smaller systems. This is why an emerging, disruptive switching technology is probably required before a viable system can be manufactured.

On the software side, besides simulation, there is a lot of other work to be done in the development environment, including:

- development of a display system for working with three-dimensional circuits — possibly something that uses an immersive environment;
- integration of the layout tools with the simulation tools, so that debugging can make reference to the original designs that were compiled;
- development of better de-compilers, which are very useful in debugging circuits that

have been generated by other circuits;

- better tools for tracing wire routing; and
- the implementation of *functional blocks*, i.e., the ability to encode higher-level functions into modules, and have those modules simulated at the higher level, rather than at the level of single reconfigurable elements.

It would also be useful to encapsulate things like fault handling into the tools themselves. For example, when designing a circuit, being able to specify what form/degree of fault tolerance is required, and having the compiler automatically augment the circuit as needed. This is part of a more-general task, which is the development of a language for representing self-configuring circuitry. One proposed method is the use of something called "magic polygons," that are basically self-modifying schematics [**?**].

The proposed architecture embodies the four original hypothesis stated in Chapter 1, and is based on conventional digital logic. However, there are other possible embodiments of those hypothesis that don't extend digital circuitry. One would be an *analog* version of the proposed reconfigurable system. Such a system would employ voltage levels within a given range, rather than just at the high and low ends. The truth tables within each element would not be binary input-to-output mappings, but rather a collection of component values for a network of discrete components (resistors, capacitors, inductors), as well as perhaps things like op amps. The internal-configuration process in this case would consist of shifting out/shifting in these component values to/from neighboring elements. While the specific

meaning of signal values would vary greatly from the digital case, the basic constructs — such as element replication and parallel configuration — would be essentially unchanged.