ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

PHD THESIS

A SELF-REPAIRING MULTIPLEXER-BASED FPGA INSPIRED BY BIOLOGICAL PROCESSES

GIANLUCA TEMPESTI
LOGIC SYSTEMS LABORATORY
COMPUTER SCIENCE DEPARTMENT
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
LAUSANNE, SWITZERLAND

Thesis Director:	Prof. Daniel MANGE, EPFL
Thesis Committee:	Dr. Miron ABRAMOVICI, Lucent Technologies
	Prof. Giovanni CORAY, EPFL
	Dr. Pierre KUONEN, EPFL
	Prof. MariaGiovanna SAMI, Politecnico di Milano

JULY 8, 1998

ABSTRACT

Special-purpose parallel systems, and in particular cellular SIMD (Single-Instruction Multiple-Data-Stream) and SPMD (Single-Program Multiple-Data-Stream) array processors are very interesting approaches for handling many computationally-intensive applications. These systems consist of an array of identical processing elements executing the same operations (at the instruction or at the program level, respectively) on different sets of data.

The main obstacles to the widespread use of application-specific arrays of processors are, of course, development time and price: the time required for the design of such systems is usually measured in months, if not years, while the cost of custom VLSI circuits makes such designs too expensive for most situations.

A major goal of the Embryonics project, the research project which provides the framework for this thesis, is the development of such an array of processors, or cells, inspired by biological cellular processes, and in particular by the embryological processes of living beings. These processors, relatively simple binary decision machines, are remarkable in that they are easily parametrizable in both size and functionality. This property derives from the use of a relatively new type of high-complexity reprogrammable circuits, commonly referred to as Field-Programmable Gate Arrays (FPGAs). These circuits provide a homogeneous surface of general-purpose logic elements which can be configured as often as desired to implement any combinational or sequential circuit (within the limits imposed by the number of available elements).

Unfortunately, commercial FPGAs are not well adapted to the implementation of arrays of processors, for the following main reasons:

- They are designed to be used in single-chip systems: while it is possible to link multiple FPGAs, the connections between the chips become a major bottleneck for the configuration, making it impossible to treat a multi-chip system as a homogeneous surface of logic elements.
- Neither their structure nor their configuration software allow to easily partition the chip(s) into an array of identical processing elements.
- They are not currently capable of self-repair, a required feature for very large systems (such as a sizable array of processing elements), where the probability of faults increases to the point that they become impossible to ignore. One of the main sources of inspiration

of the Embryonics project is the remarkable robustness of biological organisms.

The main goal of the thesis was therefore to design a novel type of FPGA, known as MuxTree, drawing inspiration from the world of biological organism so as to render it capable of built-in self-test and self-repair, as well as of being easily configured as an array of identical processors (the cells).

The configuration of large arrays of programmable logic elements of variable size and shape with a repetitive pattern of identical processing elements bears a strong resemblance to the problem of self-replication in cellular automata. We thus exploited the knowledge base provided by the latter domain to develop an approach which could be applied to the former. The first result of this research was the development of a novel self-replicating cellular automaton, loosely based on existing solutions. The self-replication scheme used in the automaton to perform self-replication was then simplified so as to be easily implemented in hardware and, appropriately modified, was used to direct the configuration of the array of MuxTree elements, allowing an easy partitioning into blocks of elements. The size of these blocks, and thus the size of the processors, can thus be programmed by the user to fit a particular application.

The second main research topic in this thesis concerns the development of self-test and self-repair logic for the MuxTree element. The main challenge in this area was implementing the desired features with a minimal amount of logic, while respecting a set of rigid constraints imposed by our desire to follow a biological inspiration. In the design of the self-test mechanism, we adopted a hybrid approach which operates partly on-line (i.e., while the circuit is functioning) and partly off-line (while the circuit is being configured), but always transparently to the user. To achieve self-repair, we implemented a relatively simple reconfiguration mechanism which automatically replaces the faulty elements and reroutes the array's connections. While the reconfiguration process is not always completely transparent, our mechanism does allow the circuit to resume operation without losing its internal state.

The initial goals were, for the most part, met and the circuit was implemented in the form of a set of demonstration modules (the Biodules 603). Using these modules, we were able to construct a small prototype and demonstrate our FPGA's ability to self-replicate its configuration and to self-repair in the presence of faults.

RÉSUMÉ

Les systèmes parallèles spécialisés, et en particulier les réseaux de processeurs cellulaires SIMD (Single Instruction Multiple Data) et SPMD (Single Program Multiple Data) sont des approches très intéressantes pour beaucoup d'applications gourmandes en calcul. Ces systèmes sont composés d'une matrice de processeurs exécutant les même opérations (au niveau des instructions ou du programme, respectivement) sur des données différentes.

Les obstacles principaux à une généralisation de l'usage des réseaux de ce type sont, évidemment, le temps de développement et le prix: le temps nécessaire pour concevoir de tels systèmes se mesure en général en mois, voire en année, tandis que les coûts de circuits VLSI spécialisés les rendent trop coûteux pour la plupart des applications.

Un des buts principaux du projet Embryonique, le cadre dans lequel cette thèse s'inscrit, est le développement de tels réseaux de processeurs, ou cellules, inspiré par les principes de la biologie moléculaire, et en particulier par ceux de l'embryologie des êtres vivants. Tant la taille que la fonctionnalité de ces processeurs, des machines de décision binaire relativement simples, peuvent être dimensionnées de manière remarquablement simple. Cette propriété vient de l'utilisation d'un type relativement nouveau de réseaux logiques reprogrammables à haute complexité, appelés FPGAs (Field-Programmable Gate Arrays). Ces réseaux offrent une surface homogène d'unités logiques universelles qui peuvent être reconfigurées à volonté de manière à implémenter n'importe quel circuit combinatoire ou séquentiel (dans la limite du nombre d'unités disponibles).

Malheureusement, les FPGAs commerciaux ne sont pas bien adaptés à l'implémentation de réseaux de processeurs pour les raisons suivantes:

- ils sont conçus pour être utilisés dans des systèmes qui contiennent une seule puce programmable; bien qu'il soit possible de relier plusieurs FPGAs entre eux, ces connexions deviennent vite des goulots d'étranglement, rendant impossible de les traiter comme une surface homogène d'unités logiques;
- ni leur structure, ni leur logiciel de configuration ne permettent de partitionner facilement la(les) puce(s) en réseaux d'unités de traitement identiques;
- ils ne sont pas encore capables d'autoréparation, une propriété nécessaire pour de très grands systèmes (tels que des réseaux de processeurs), où la probabilité de fautes augmente au point qu'il devient impossible de les ignorer; une des principales sources d'ins-

piration du projet Embryonique est précisément la remarquable robustesse des organismes biologiques.

Le principal objectif de cette thèse peut donc se résumer à la conception d'un nouveau type de FPGA, appelé MuxTree, inspiré par la biologie. Plus particulièrement, ce sont les qualités d'autoréparation et d'autotest, inspirées par les organismes vivants, ainsi que la capacité à être facilement décomposé en réseaux de processeurs (les cellules) qui ont motivé la conception de ce nouveau FPGA.

Dans ce travail, nous avons exploité les connaissances existantes dans le domaine des automates cellulaires autoréplicateurs pour développer des techniques applicables aux grands réseaux logiques reprogrammables. En effet, la configuration de ceux-ci, structurés en éléments de forme et de taille régulières, ressemble beaucoup au problème de l'autoréplication dans les réseaux cellulaires.

Ce premier axe de recherche a conduit au développement d'un nouveau type d'automates cellulaires autoréplicateurs partiellement inspirés de solutions déjà existantes. La stratégie d'implémentation de l'autoréplication sur cet automate a été simplifiée au maximum de manière à l'adapter aisément sur le matériel. Cette stratégie a été ensuite appliquée aux FPGAs pour permettre la configuration de réseaux cellulaires matériels. Ce processus, appliqué sur une structure régulière composées d'éléments MuxTree, permet de partitionner très simplement cette dernière en blocs. La taille de ceux-ci et, par conséquent, la taille des processeurs réalisés, est ajustable en fonction de l'application visée.

Le deuxième axe de recherche dans cette thèse est le développement d'une logique embarquée pour l'autotest et l'autoréparation d'éléments MuxTree. Un défi important a été imposé par le désir de minimiser la quantité de logique nécessaire pour ces fonctions tout en respectant les contraintes de l'inspiration biologique. Le mécanisme d'autotest que nous avons mis au point repose sur une approche hybride, invisible pour l'utilisateur, qui combine le test en ligne (pendant le fonctionnement de l'application) et le test hors ligne (pendant le chargement de l'application). Quant au mécanisme d'autoréparation, nous avons conçu et implémenté une stratégie simple basée sur le remplacement des éléments fautifs et le reroutage de signaux du réseau. Bien que cette méthode ne soit pas complètement transparente, elle permet néanmoins de reprendre l'exécution normale de l'application sans perdre l'état interne.

Les objectifs initiaux ont été en grande partie atteints et le circuit a été implémenté sous la forme de modules de démonstration (Biodule 603). Ces derniers nous ont permis de construire un prototype réduit et de démontrer les capacités de notre FPGA à s'autorépliquer et à s'autoréparer.

ACKNOWLEDGMENTS

A Ph.D. thesis is the culmination of years of study. Throughout these years, I have received much support from family, teachers, friends. It would be impossible to individually thank of all those who deserve it. I *will*, however, acknowledge my debt to those people who made this thesis possible and enriched my life in the past few years.

First,of course, my parents and family, who have always supported me and have been willing to make considerable sacrifices in order to give me all possible advantages in life. Needless to say, without them I would not be where I am today.

Daniel Mange, much more than a thesis director: a mentor and a friend. He is the true driving force of the Embryonics project, and he has taught me much, both directly and by providing a shining example.

The group of experts who shouldered the (hopefully not too heavy) burden of reading and appraising my work: Miron Abramovici, Giovanni Coray, Pierre Kuonen, and Mariagiovanna Sami, as well as Jean-Daniel Nicoud, head of the department. My heartfelt thanks for their precious feedback and for their favorable evaluation.

The "ontogenetic team", the exceptional group of people with whom I had the extreme pleasure to work: André Stauffer, Jacques Zahnd, and our colleagues at CSEM, Pierre Marchal, Pascal Nussbaum, and Christian Piguet. If this thesis is worth something, it is also thanks to them.

Marlyse Taric, our secretary, always there for us all. It remains a mystery to me how she manages to do all she does while working in our lab only part-time.

I will take this opportunity to thank all those members of the laboratory who have contributed to make my stay so pleasant. I dislike unnecessary seriousness and thus my remarks will be (hopefully) humorous. But do not let my glibness mislead you: I am sincerely grateful to all these people for making the past few years a thoroughly enjoyable and rewarding experience.

To begin with, those with whom I kindly condescend to share my office: Emeka Mosanya, beer-guzzler extraordinaire, to whom I owe the redoubtable experience of tasting a *mitraillette* (trust me, you don't want to know...); Dominik Madon, resident political activist, office DJ, and yearly winner of the lab's "best-looking hair" competition; Mathieu Capcarrère, whose low resistance to alcohol is more than compensated by his willingness to retrieve lake-bound volleyballs. I can now proudly claim that, all their efforts notwithstanding, I *still* managed to finish my thesis.

André “Chico” Badertscher, wizard of the soldering iron, heroic glider pilot, and unchallenged champion of the truly execrable pun. His contributions to this thesis are extensive: to his outstanding technical skills and ability to work under stress goes the credit for assembling the prototype, and, more importantly, to his punctual restocking of the lab’s cafeteria I owe the caffeine required for any successful thesis.

The Colombian armada, from its *lider maximo*, Eduardo Sanchez, to its brave lieutenants, Carlos Andrés Peña, Andrés Perez-Uribe, and Héctor Fabio Restrepo, not to forget its honorary member, Jean-Michel Puiatti (who, Italian origins notwithstanding, is probably the greatest hispanophile of them all). Through their influence, Spanish (well... Colombian) will soon become the official language of the LSL.

The boys from Valais: Jacques-Olivier “Jacko” Haenni, the omnipotent (if far-from-omniscient) sysadmin, and Jean-Luc Beuchat, he of the doubtful musical taste.

Moshe Sipper, keeper of the list of references (which has now reached such gargantuan proportions that it is likely to soon achieve critical mass) and official lab trekkie.

Finally, some oldies but goodies: Serge Durand, who introduced me to the joys of Embryonics, before leaving our beautiful ivory tower for the crass and material world of industrial R&D; Christian Iseli, the much-missed ex-sysadmin, and his family, whose presence brought considerable surges of random noise and frantic activity; Marco Tomassini, who single-handedly doubled the lab’s Italian contingent during his all-too-brief stay.

This work was partially sponsored by grants 20-39391.93 and 20-42270.94 from the Swiss National Science Foundation. In a time when funding is more and more often granted only to projects with immediate industrial applications, it is heartening to know that long-term research is still possible.

TABLE OF CONTENTS

Chapter 3 Self-Replication

Chapter 4 Self-Repair

CHAPTER 1

INTRODUCTION

Biological organisms are among the most intricate structures known to man, exhibiting highly complex behavior through the massively parallel cooperation of huge numbers of relatively simple elements, the cells. As the development of computing systems approaches levels of complexity such that their synthesis begins to push the limits of human intelligence, more and more engineers are beginning to look at nature to find inspiration for the design of computing systems, both in software and in hardware. This thesis will present one such endeavor, notably an attempt to draw inspiration from biology in order to design a novel digital circuit, endowed with a set of features motivated and guided by the behavior of biological systems: *self-replication* and *self-repair*.

In this introductory chapter, we will present a short description of the motivations behind the development of our circuit, that is, the reasons which led us to draw inspiration from biology in our design (section 1.1). We will then introduce the basic features we wish to introduce in our new circuit (section 1.2), and conclude the chapter with a brief outline of the overall structure of the thesis (section 1.3), including an overview of our original contributions.

1.1 Motivations

Biological inspiration in the design of artificial machines is not a new concept: the idea of robots and mechanical automata as man-like artificial creatures by far predates the development of the first computers. With the advent of electronics, the attempts to imitate biological systems in computing machines did not stop, even if their focus shifted from the mechanical world to the realm of information: since the physical substrate of electronic machines (i.e., the hardware) is not easily modifiable, biological inspiration was applied almost exclusively to information (i.e., the software), albeit with some notable exceptions [29].

Recent technological advances, in the form of programmable logic circuits [14, 103], have engendered a re-evaluation of biological inspiration in the design of computer hardware. This thesis, and the larger project which encompasses it, represent an attempt at exploiting such an inspiration in the design of digital circuits. This section contains a brief overview of the foundations of the work presented in this thesis.

1.1.1 Von Neumann's Universal Constructor

The field of bio-inspired digital hardware was pioneered by John von Neumann [10]. A gifted mathematician and one of the leading figures in the development of the field of computer engineering, von Neumann dedicated the final years of his life on what he called the *theory of automata* [104]. His research, which was unfortunately interrupted by von Neumann's untimely death in 1957, was inspired by the parallel between *artificial automata*, of which the paramount example are computers, and *natural automata* such as the nervous system, evolving organisms, etc.

Through his theory of automata, von Neumann conceived of a set of machines capable of many of the same feats as biological systems: evolution, learning, self-replication, self-repair, etc. At the core of his approach was the development of *self-replicating machines*, that is, machines capable of producing identical copies of themselves. In developing his theory, Von Neumann identified a set of criteria which had to be met in order to obtain useful biological-like behavior in computing machines. These criteria rested on two fundamental assumptions:

- Self-replication should be a special case of *construction universality*. That is, the self-replicating machines should be able not only to create copies of themselves, but also to construct any other machine, given its description. Such a feature would be a requirement to later obtain evolving systems.
- The self-replicating machines should be *universal computers*, that is, capable of executing any finite (but arbitrarily large) program. Such machines were known to von Neumann: the requirement of logical universality is met by a class of automata known as *universal Turing machines* [39, 55].

Von Neumann's research was, as we mentioned, never completed: the only machine he developed to any great extent was a theoretical model known as the *universal constructor*. Nevertheless, his theory of automata provides even today a firm foundation for the development of bio-inspired systems.

1.1.2 Field-Programmable Gate Arrays

Von Neumann's universal constructor was probably the first example of self-replicating computer hardware. Unfortunately, electronic technology in the fifties did not allow the development of a machine so complex. As a consequence, research concerning hardware self-replication waned for a number of years.

In the eighties, bio-inspiration gained new momentum under the label of *artificial life*, a research field pioneered by Christopher Langton which is attracting more and more interest in the scientific community. Under the impulse of new technology, bio-inspired hardware is also finally reaching the stage of physical realization.

The key technology which today allows the development of such approaches is the advent of programmable logic devices, usually referred to as *field-programmable gate arrays* (FPGAs) [14, 103]. These devices consist of two-dimensional arrays of identical elements. Each of these elements is designed to be able to implement a variety of different functions, depending on the value of its *configuration*, a string of bits defined by the user at run-time. The size of an FPGA element (known as its *grain*) can vary considerably from one device to the next, ranging from complex look-up table based architectures (*coarse* grain) to much smaller hard-wired elements (*fine* grain). The elements are connected to each other through a connection network which is itself programmable.

A hardware designer can use an FPGA to implement just about any kind of digital logic circuit by defining the functionality of each element as well as the connections between these elements at run-time. Therefore, they are the ideal platform for the development of bio-inspired hardware, which requires that the layout of the circuit be modified through mechanisms such as self-replication, evolution, or healing (self-repair).

The goal of the work presented in this thesis is to develop an FPGA architecture which exploits biologically-inspired mechanisms to introduce two novel features: self-replication and self-repair. The resulting circuit, a very fine-grained FPGA known as MuxTreeSR (for *tree of multiplexers with self-repair*), was designed with a specific application in mind: its use as a platform for the implementation of the more complex bio-inspired structures we developed in the framework of a larger project, known as *Embryonics*.

1.1.3 Embryonics

This thesis is part of a more general research project, called Embryonics [60] (a contraction of the words *embryonic electronics*), which aims at establishing a bridge between the world of biology and that of electronics, and in particular between biological organisms and digital circuits.

As the possible intersections between these two worlds are manifold, so the Embryonics project advances along more than one research axis, investigating domains as diverse as artificial neural networks [36, 74] and evolutionary algorithms [64, 99].

The research axis to which this thesis belongs is concerned with the use of biologically-inspired mechanisms in the synthesis of digital circuits, and draws inspiration from two distinct sources. The first is the biological mechanism of multi-cellular organization: the complex behavior of natural organisms derives from the parallel operation of a multitude of simple elements, the cells, each containing the complete description of the organism (the *genome*). The second is von Neumann's concept of self-replication of an universal computer, a mechanism which allows for the automatic creation of multiple identical copies of a machine from a single initial copy.

These two approaches are fundamentally different. Both rely on a mechanism of self-replication to obtain arrays of elements which can be seen as processors, all executing an identical program. However, in von Neumann's case the processors are universal Turing machines, and are identical in structure as well as in functionality: the process of cellular differentiation is entirely absent, and the whole system can be seen as a self-replicating *unicellular* organism. In nature, cells are different in structure and functionality (the appearance and behavior of a liver cell, for example, are considerably different from that of an epidermal cell), but any cell is potentially capable of replacing any other cell because it contains the description of the entire organism, i.e., the genome. Cellular differentiation is therefore at the very core of biological systems, which derive their complexity from the coordinated operation of small, differentiated cells.

In Embryonics, we developed a solution which tries to integrate the two approaches: our system consists of an array of artificial cells implemented by small processors which have an identical structure (the same hardware layout) but different functionality (different software). The biological inspiration is introduced by storing in each processor the code required by the entire array: each artificial cell will then select, depending on its position within the array, which portion of code to execute. The functionality of each processor will therefore be different, as in biology, because it executes a different part of its artificial genome. This considerable redundancy goes against conventional design rules (which emphasize the minimization of the size of the elements), but proves useful in the implementation, for example, of self-repair: since each cell contains the same code as any other cell in the system, then it can also *replace* any cell simply by executing a different portion of the artificial genome.

The approach used in the Embryonics project is thus based on the parallel operation of an array of artificial cells, each consisting of a small processor and containing the information required by the entire system. The novel FPGA presented in this thesis was designed so as to implement the self-replication of our cells directly in hardware, thus allowing the user to automatically realize the cellular array staring from the description of a single artificial cell. The programmable logic of our circuit can thus be seen as a *molecular layer*, providing the supporting material for the construction of our artificial cells.

Obviously, our array of processors will require a considerable amount of programmable logic, which implies that the probability of faults occurring somewhere in the circuit will be non-negligible. Hence the need to implement a mechanism which allows the circuit to operate in the presence of faults. Biological systems are faced with the same requirement, and solve it by replacing dead cells with new ones (*cicatrization*). Since the creation of new silicon is not feasible given current technology, the electronic equivalent of cicatrization (i.e., self-repair) will have to rely on the presence of spare logic which can replace the faulty part of the circuit. Our FPGA will provide support for self-repair directly in hardware, thus simplifying the task of developing self-healing systems.

1.2 Features

In order to design an FPGA tailored for the requirements of the Embryonics project, we thus needed to introduce two biologically-inspired features: self-replication and self-repair.

The first feature is directly related to von Neumann's approach and is at the core of biological inspiration in Embryonics, since it allows the creation of arrays of artificial cells starting from the description of a single such cell. In order to achieve a physical realization of von Neumann's machine, that is, the self-replication of an universal computer, our system will require the following capabilities:

- It should be able to construct multiple copies of any machine from the description of a single such machine. Ideally, it should be the machines themselves to generate and direct the copying process.
- The process should be applicable to machines capable of executing any given task.
- As a corollary to the above, the process should be applicable to machines of any given size.

Our task in designing a self-replicating FPGA should therefore be fairly obvious: since our artificial cells are universal machines capable of executing any given task given a sufficiently large memory (and are thus universal computers), in order to fulfill the requirements laid out by von Neumann our FPGA will require a mechanism capable of constructing multiple copies from the description of a single cell.

Self-repair depends on a somewhat different set of assumptions, related more to engineering than to biology. In biology, as well as in von Neumann's approach, self-repair is achieved through the replacement of faulty *cells*. As we will see in the next chapter, such a mechanism is indeed present in our machines: faulty cells are replaced by identical spare cells whenever necessary.

Introducing self-repair in our FPGA is the equivalent of cicatrization at the *molecular* level, a phenomenon which has no direct parallel in either biology or in von Neumann's work. Nevertheless, from an engineer's standpoint, it is a mechanism which is extremely interesting for a number of reasons:

- A two-level self-repair system is likely to be more versatile and powerful than a single-level one. Self-repair at the molecular level implies that we will not need to sacrifice an entire cell (which, being a processor, is likely to occupy a considerable amount of programmable logic) for every single fault occurring in the system.
- Ideally, self-repair should be transparent to the user, occurring while the circuit is operating. Such a requirement is more likely to be fulfilled through dedicated hardware, and therefore at the molecular level.

- Self-repair mechanisms, and particularly the self-test mechanisms involved in fault detection, are very much dependent on the structure of the circuit being repaired. If most or all faults are detected at the molecular level, then achieving self-test at the cellular level becomes much simpler[1].

From these observations, we can begin to outline the basic feature of our ideal self-repair mechanism. First of all, of course, it will require a self-test mechanism capable to detect as many faults as possible (ideally, it should be able to detect *all* possible faults in the system, but such a goal is not likely to be achievable) transparently to the user. Moreover, such a system should be able to accurately determine the exact location of the fault (that is, which of the elements in the array is faulty) so that self-repair can restore the functionality of the circuit. As far the self-repair mechanism itself is concerned, it should be able to repair as many faults as possible (once again, it is not reasonable to assume that all faults will be repairable) and, should such a repair not be possible at the molecular level, activate the self-repair process at the cellular level.

Obviously, these features will introduce both additional hardware and additional delays in our circuit. Since our FPGA is extremely fine-grained, it will be very important to minimize the hardware overhead. On the other hand, the speed of operation is not an important factor in our research (biological systems, after all, operate relatively slowly). Our main effort in this project was therefore to minimize space (area) rather than time (delay), while of course respecting the considerable number of additional constraints imposed by the biological inspiration of our system.

1.3 Outline

The mechanisms we developed to implement self-replication and self-repair are not completely independent: as we will see, certain features of the self-replication mechanism can be justified only as a consequence of the self-repair mechanism, and vice-versa. A linear description of the entire system is therefore extremely difficult. Nevertheless, we have tried to keep the two systems as separate as possible, and the overall structure of this document reflects such an attempt.

Many of the choices in the design of our system stem from its being part of a larger project. Therefore, in chapter 2 we will provide an overview of the Embryonics project as a whole, as well as introduce some examples of biological inspiration in computer science, greatly expanding the brief outline provided above in section 1.1. The Embryonics project is, of course, a collective effort. As such, we cannot claim that the contents of this chapter are original work, even if we hope

1. An important advantage: since the structure of our artificial cells depends on the application, the self-test mechanism at the cellular level might also need to be altered for each application.

to have contributed to some extent in the project's development. However, we feel that such an introduction is necessary to fully understand the motivations of our design.

Chapter 3 is dedicated to self-replication. The development of a mechanism implementing this feature required a considerable amount of original research. In order to introduce the problem, the chapter will begin with an analysis of the existing approaches to self-replication in computer science, including a description of von Neumann's seminal work on the subject. After the historical background, we will introduce the original research which led to the final design of a self-replication mechanism for our FPGA.

Chapter 4 will contain a description of our self-repair mechanism. It will start with a description of the FPGA in its basic form, that is, *without* self-replication and self-repair. Next, we will introduce the problem of self-test and its implementation, before a description of the self-repair mechanism. The latter will also include a description of the modifications to the self-replication mechanism required by self-repair. Neither the self-test nor the self-repair mechanism can be considered original work by themselves, relying as they do on relatively well-known approaches. The originality of the material described in this chapter lies more in the implementation: applying these features to a fine-grained FPGA required a major effort to minimize the hardware overhead, which in turn required a careful analysis and simplification of the existing approaches. The chapter will then close with a simple but complete example of the operation of our system.

In the conclusion (chapter 5), we will analyze our system with respect to the initial requirements outlined above in section 1.2, and investigate possible applications for our system outside of the Embryonics project. The main body of the project will then end with a few considerations on possible future developments for the Embryonics project in general and for our FPGA in particular.

The body of the thesis will be followed by two annexes: the first (Annex A) will describe a software package we developed in order to help us in the design of our self-replication mechanism; the second (Annex B) will describe in detail a prototype of our FPGA which we designed in order to demonstrate the feasibility of our system.

CHAPTER 2

THE EMBRYONICS PROJECT

The objective of the work presented in this thesis is the development of a programmable circuit (FPGA) which will integrate some properties which are more commonly associated with biological organisms: self-repair (healing) and self-replication (cellular division). We try to achieve this goal by applying bio-inspired mechanisms to digital electronics, an approach which provides a set of useful guidelines for the development of our system, but also introduces some peculiar constraints which are not commonly encountered in traditional circuit design.

In order to understand some of the choices we made in our design, it is therefore necessary to consider the biological inspiration of our project. This section aims at providing the required background by presenting an overview of the Embryonics project and of biological inspiration (section 2.1) and then narrowing the focus on the more specific features of bio-inspired hardware (section 2.2).

2.1 Biological Inspiration in Embryonics

In this section we want to provide an overview of the approach we adopted in the Embryonics project in order to adapt biological concepts and mechanisms to the world of electronics. After introducing the project and its main goals, we will present our approach to biological inspiration, and notably the model we created to try and unify bio-inspired systems into a common framework.

2.1.1 Overview of the Project

Embryonics [60, 62, 89, 94] is a long-term research project, conceived not so much to achieve a specific goal, but rather to look for insights by applying new concepts to a known field. In our case, we try to obtain interesting results by applying biological concepts (i.e., concepts which are usually associated with biological processes) to computing, and notably to the design of digital hardware.

The analogy between biology and electronics is not as farfetched as it might appear at a first glance. Aside from the more immediate parallel between the human brain and the computer, which has led to the development of fields such as artificial intelligence and is the inspiration for many applications (such as, for example, pattern recognition), a certain degree of similarity exists between the *genome* (the hereditary information of an organism) and a computer program.

The genome consists of a uni-dimensional string of data encoded in a base-4 system. The DNA (Deoxyribonucleic Acid), the macromolecule in which the genome is encoded, is a sequence of four bases: A (Adenine), C (Cytosine), G (Guanine), and T (Thymine). The information stored on the DNA is chemically decoded and interpreted to determine the function of a cell. A computer program is a unidimensional string of data encoded in a base-2 system (0 and 1). Stored in an electronic memory circuit, it is interpreted to determine the function of a processor.

Of course, carbon-based biology and silicon-based computing are different enough that no straightforward one-to-one relationship between the genome and a computer program (and indeed between any biological and computing process) can be established, except at a very superficial level. However, through careful interpretation, some basic biological concepts can be adapted to the design of computer systems, and some biological processes are indeed extremely interesting from a computer designer's point of view: for example, an organism's robustness (achieved by its healing processes) is unequaled in electronic circuits, while the natural process of evolution has produced organisms of a complexity which far exceeds that of modern computer systems.

It should be obvious that, biology being a very vast field, the biological concepts which could potentially be adapted to computer hardware are numerous, as are the approaches to applying such concepts to electronics. The first step in such an effort should therefore be to identify which aspects of the biological world will be studied, and define a unified framework which can meld them into a single project.

2.1.2 Bio-Inspired Systems and the POE Model

In recent years, the barrier dividing the worlds of biology and electronics has considerably thinned. More and more, electronics are exploited in biology and medicine, while biological ideas are starting to cross over into computer design.

Even if the first direction is, so far, definitely dominant (computers are quickly becoming indispensable tools for medical and biological applications ranging from computerized axial tomography to genomic sequence analysis), biology is starting to inspire engineers to develop quasi-biological systems. Among the best-known examples of such systems, we can mention:

- *Expert systems* [44], very complex software programs which try to imitate the high-level processes of human intelligence through the use of dedicated algorithms. A practical application of artificial intelligence, they have produced remarkable results in a variety of fields.
- *Neural networks* [9, 36], arrays of small processing elements directly inspired by the human nervous system (more below), capable of "learning", i.e. of modifying their own structure in response to a set of input patterns, somewhat analogously to the way synapses are modified by external (sensorial) stimuli.

- *Genetic algorithms* [65] and *evolutionary computation* [64], which exploit some of the mechanisms of natural evolution (such as mutations and sexual reproduction) in the hope of evolving a solution for computational problems which are either too complex or too ill-defined to be solved with conventional approaches.
- *Computer viruses* [91], tiny (but extremely crafty) computer programs which attempt to invade and sometimes destroy a computer system using strategies very similar to those used by biological viruses to invade and sometimes kill an organism [46].

Obviously, biological inspiration in computer science can assume a variety of different forms, and Embryonics does not attempt to cover all its possible aspects. Nevertheless, the scope of the project is such that it does involve many of the traditional bio-inspired systems, as well as the development of novel approaches. To illustrate the scope of the project, we have created the POE model [84, 90], an attempt to classify bio-inspired systems, according to their biological source of inspiration, along three axes: phylogeny, ontogeny, and epigenesis (Fig. 2-1).

2.1.3 The POE Model: Phylogeny

> *Phylogeny*, n.: The racial history or evolutionary development of any plant or animal species [107].

On the phylogenetic axis we find systems inspired by the processes involved in the evolution of a species through time, i.e. the evolution of the genome. The process of evolution is based on alterations to the genetic information of a species, occurring through two basic mechanisms: *crossover* and *mutation*.

Crossover (or recombination) is directly related to the process of sexual reproduction. When two organisms of the same species reproduce, the offspring contains genetic material coming from both parents, and thus becomes a unique individual, different from either parent. Mutation consists of random alterations to the genome caused either by external phenomena (i.e., radiation or cosmic rays) or by chemical faults which occur when the two genomes merge. Often resulting in non-viable offspring[1], mutations are nevertheless vital for the process of evolution, as they allow "leaps" in evolution which would be impossible to achieve by merging the genomes of individuals of the same species.

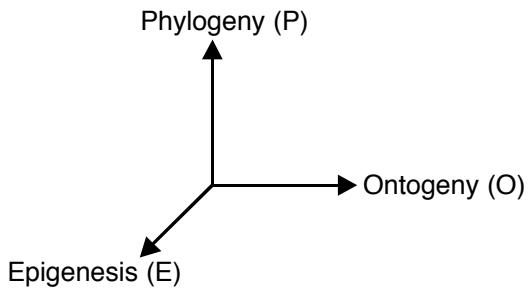

Figure 2-1: The POE Model.

1. In fact, the great majority of mutations have no effect, as they act on unused parts of the genome.

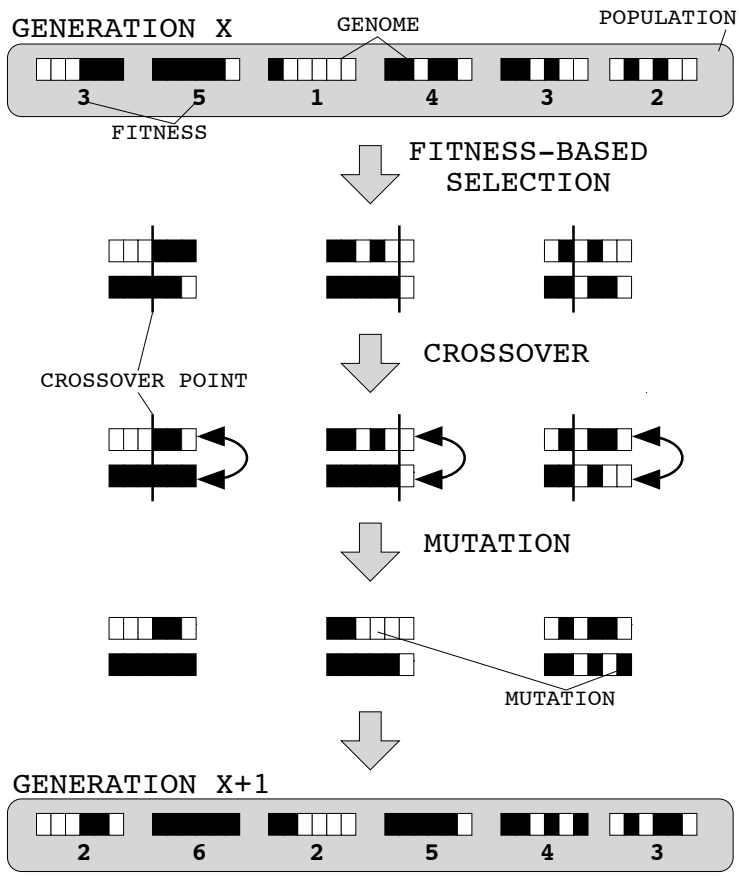

Figure 2-2: Example of a single iteration of an evolutionary algorithm applied to a problem for which the fitness consists of maximizing the number of black tiles.

Both these mechanisms are, by nature, non-deterministic. This represents both their strength and their weakness, when applied to the world of electronics. It is a strength, because they are fundamentally different from traditional algorithms and thus are potentially capable of solving problem which are intractable by deterministic approaches. It is a weakness, because computers are inherently deterministic (it is very difficult, for example, to generate a truly random number, a basic requirement for non-deterministic computation, in a computer).

Even with this disadvantage, algorithms which exploit phylogenetic mechanisms are carving themselves a niche in the world of computing. These algorithms, commonly referred to as *evolutionary algorithms* (Fig. 2-2) (a label which regroups domains such as genetic algorithms [65], evolutionary programming [28], and genetic programming [48, 49]), are usually applied to problems which are either too ill-defined or intractable by deterministic approaches, and whose solutions can be represented as a finite string of symbols (which thus becomes the equivalent of the biological genome). An initial, random population of individuals (i.e., of genomes), each representing a possible solution to the problem, is iteratively "evolved" through the application of mutation (i.e., random alterations of a sequence) and crossover (i.e., random mergings of two sequences). The resulting

sequences are then evaluated on the basis of their efficiency in solving the given problem (the fitness function) and the best (fittest) solutions are in turn evolved. This approach is not guaranteed to find the best possible solution to a given problem, but can often find an "acceptable" solution more efficiently than deterministic approaches. Our laboratory is very active in this domain [86, 87, 88].

Another, less conventional but very interesting example of phylogenetic computer system is Tierra [81], an experimental project in which a "population" of programs is allowed to freely evolve inside a computer's memory. The very simple fitness criterion is survival in an environment with limited resources (in this case, the limited resource is the amount of memory available). Some very interesting and unexpected survival strategies emerged spontaneously, including parasitism (where tiny programs, which would not be able to survive by themselves, exploit the code belonging to other programs) and the evolution more and more compact programs (smaller programs are more likely to survive in an environment where memory is the limited resource).

It appears, then, that the phylogenetic axis has already provided a considerable amount of inspiration to the development of computer systems. To date, however, its impact has been felt mostly in the development of software algorithms, and only marginally in the conception of digital hardware. Koza et al. pioneered the attempt to apply evolutionary strategies to the synthesis of electronic circuits when they applied genetic algorithms to the evolution of a three-variable multiplexer and of a two-bit adder [50]. Also, evolutionary strategies have been applied to the development of the control circuits for autonomous robots [26, 27]. Unfortunately, technical issues have posed severe obstacles to the progress of this kind of approach, and few groups are currently active in this domain [37, 38, 100]. Our laboratory contributed to the research with the development of Firefly [33], a machine capable of evolving a solution to the problem of synchronizing a unidimensional cellular automaton (see the next chapter for a more detailed description of cellular automata). This relatively simple task represents nevertheless the first example of hardware capable of evolving completely on-line, i.e. without the assistance of a computer.

2.1.4 The POE Model: Epigenesis

> *Epigenesis*, n.: The theory that the germ cell is structureless and that the embryo develops as a new creation through the action of the environment on the protoplasm [107].

The human genome contains approximately $3x10^9$ bases. An adult human body contains something like $6x10^{13}$ cells, of which approximately 10^{10} are neurons, with 10^{14} connections. Obviously, the genome cannot contain enough information to completely describe all the cells and synaptic connections of an adult organism [21].

There must therefore exist a process which allows the organism to increase in complexity as it develops. Since we already know that the additional information

cannot come from within the cell itself, the only possible source of additional information is the outside world: the growth and development of an organism must be influenced by the environment, a process which is most evident in the development of the nervous, immune, and endocrine systems. Following A. Danchin's terminology [21, 22], we have labeled this process *epigenesis*[2].

Epigenetic mechanisms have already had considerable impact on computer science, and particularly on software design, notably through the concept of *learning*. The parallel between a computer and a human brain dates to the very earliest days of the development of computing machines, and led to the development of the field known as *artificial intelligence* [108].

AI, which probably knew its greatest popularity in the seventies, tried to imitate the high-level processes of the human mind using heuristic approaches. After a number of years in the spotlight, this field seems today to have lost momentum, but not without leading to useful applications (such as expert systems) and major contributions to the development of novel algorithms (for example, IBM's well-known Deep Blue computer exploits algorithms largely derived from AI research).

Hardware systems based on epigenetic processes are starting to emerge, in the form of artificial neural networks (ANNs) [9, 36], two-dimensional arrays of processing elements (the neural cells) interconnected in a relatively complex pattern (Fig. 2-3). Each cell receives a set of input signals from its neighbors, processes them according to a given, implementation-dependent function, and propagates the results through the network. This process is a good approximation of the mechanisms exploited by biological neurons (for example, as in nature, some inputs signals have a greater impact, or *weight*, in the computation of a neuron's output value).

The first phase in the operation of this kind of circuit is the learning phase: a set of inputs is applied to the system, the outputs are compared to a set of correct responses, and a feedback process modifies the weight of the connections and the function of the processing elements until the network has learned to "behave" correctly (i.e., to produce the correct output for most, is not all, input patterns). The feedback process is hidden from the user, who therefore cannot directly modify the weights or the functions.

ANNs cannot be expected to perform correctly (i.e. to produce the correct output) for *all* input patterns, a drawback which prevents their use in many applications. However, they have proved their worth (to the point that they are starting to be adopted in commercial systems) in applications such as voice and character recognition, where a limited margin of error is acceptable.

Our laboratory is also involved in research along the epigenetic axis, with the development of FAST (Flexible Adaptable-Size Topology) [74, 75, 76], a neural network with a dynamically reconfigurable structure. Traditional ANNs have a

2. This label is by no means universally accepted. We thank Prof. Marcello Barbieri of the University of Ferrara, Italy, for bringing this fact to our attention.

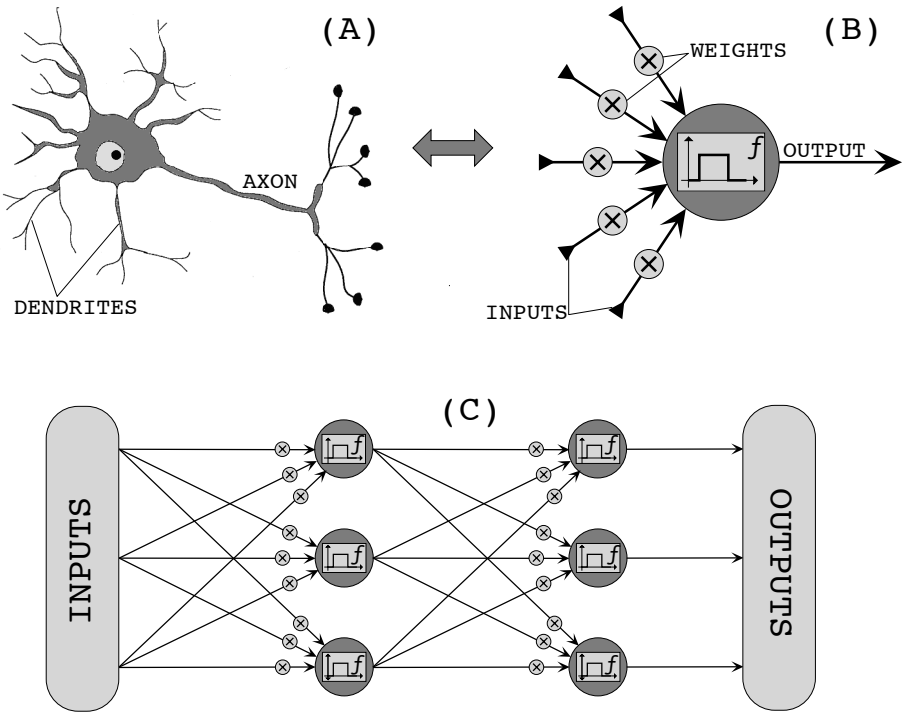

Figure 2-3: Artificial Neural Networks: (A) a biological neuron, (B) an artificial neuron, and (C) a simple network.

fixed interconnection structure, and only the weights associated with the connections can be modified. By implementing the network using reconfigurable logic (see subsection 2.2.2), FAST achieves on-line learning capabilities and can exploit an adaptable topology (two features which are essential to obtain true learning systems, as opposed to "learned" ones). While FAST is not the first neural network based on an adaptable topology [30, 67], it is unique in that it does not require intensive computation to reconfigure its structure, and can thus exist as a stand-alone machine which could be used, for example, in real-time control applications.

2.1.5 The POE Model: Ontogeny

> *Ontogeny*, n.: The life cycle of a single organism; biological development of the individual [107].

The phylogenetic and epigenetic axes of the POE model cover the great majority of existing bio-inspired systems. The development of a multi-cellular biological organism, however, involves a set of processes which do not belong to either of these two axes. These processes correspond to the growth of the organism, i.e. to the development of an organism from a mother cell (the *zygote*) to the adult phase. The zygote divides, each offspring containing a copy of the genome (*cellular division*). This process continues (each new cell divides, creating new offspring, and so on), and each newly formed cell acquires a functionality (i.e., liver cell, epidermal cell, etc.) depending on its surroundings, i.e., its position in relation to its neighbors (*cellular differentiation*).

Cellular division is therefore a key mechanism in the growth of multi-cellular organisms, impressive examples of massively parallel systems: the $6x10^{13}$ cells of a human body, each a relatively simple elements, work in parallel to accomplish extremely complex tasks (the most outstanding being, of course, intelligence). If we consider the difficulty of programming parallel computers (a difficulty which has led to a decline in the popularity of such systems), biological inspiration could provide some relevant insights on how to handle massive parallelism in silicon.

A fundamental feature of biological organisms is that each cell contains the blueprint for the entire organism (the genome), and thus can potentially assume the functionality of any other cell: no single cell is indispensable to the organism as a whole. In fact, cells are ceaselessly being created and destroyed in an organism, a mechanism at the base of one of the most interesting properties of multi-cellular organisms: healing.

The mechanisms of ontogeny (cellular division and cellular differentiation), unlike those of epigenesis and phylogeny, are completely deterministic, and are thus, in theory, more easily adaptable to the world of digital circuits (which is by nature deterministic). In spite of this, the ontogenetic axis has been almost completely ignored by computer scientists, despite a promising start in the fifties with the work of John von Neumann, who developed a theoretical model of a *universal constructor*, a machine capable of constructing any other machine, given its description [104]. Given a description of itself, the universal constructor can then self-replicate, a process somewhat analogous to cellular division.

We will describe von Neumann's machine, one of the fundamental sources of inspiration for the Embryonics project in general and for this work in particular, in detail in the next chapter, but we should mention that, unfortunately, electronic circuits in the 1950s were too primitive to allow von Neumann's machine to be realized, and the concept of self-replicating machines was thus set aside.

Probably the main obstacle to the development of self-replicating machines was the impossibility of physically creating self-replicating hardware. In fact, such machines require a means to transforming information (i.e. the description of a machine) into hardware, and such a means was definitely unpractical until recently. As we will see below, the introduction of programmable logic circuits, by demonstrating the feasibility of such a process, was an important step towards the development of self-replicating machines.

One of the main goals of the Embryonics project is to determine if, given modern technology, Von Neumann’s dream of a self-replicating machine can be realized in hardware. The work presented in this thesis represents a major step in this direction: by creating a hardware system capable of self-replication, we open the way to the practical realization of von Neumann's universal constructor.

2.2 Bio-Inspired Hardware

For a project such as Embryonics, the transition from theory to implementation is far from obvious: for every biological concept we wish to translate into hardware, we must find an equivalent concept in the world on silicon[3]. This section describes our efforts to define a parallel between the cellular structure of biological organisms and the design of computing hardware. Subsection 2.2.1 will provide a basic overview of our approach, and will be followed (subsection 2.2.2) by a short introduction to FPGAs, the programmable circuits at the heart of our system. The following subsections will then describe the three-level approach (organism, cell, molecule) we followed in the design of our bio-inspired hardware.

2.2.1 Ontogenetic Hardware

Since the ontogenetic axis, the most closely concerned by this work, is essentially based on the concept of "cell", the first step in developing ontogenetic hardware is therefore to define the electronic equivalent of a biological cell. Obviously, there exist a number of possible approaches to establish such a parallel, but if we are to maintain the analogy with biology, our electronic cell must respect the fundamental biological definitions [110]:

> *Cell*, n.: the basic structural and functional unit of all organisms; cells may exist as independent units of life (as in monads) or may form colonies or tissues as in higher plants and animals.
>
> *Nucleus*, n.: a part of the cell containing DNA and RNA and responsible for growth and reproduction.
>
> *DNA*, n.: a nucleic acid consisting of large molecules shaped like a double helix; associated with the transmission of genetic information.
>
> *RNA*, n.: a nucleic acid that transmits genetic information from DNA to the cytoplasm; controls certain chemical processes in the cell.
>
> *Genome*, n: one haploid set of chromosomes with the genes they contain; the full DNA sequence of an organism.

In other terms, the cell is the smallest structural unit of an organism which contains the description of the entire organism, i.e., its genome, which is interpreted to control the cell's functions.

One approach which allows us to respect these definitions is to implement computing systems (the artificial organisms) using an array of (relatively) small processing elements (the artificial cells), each executing the same program (the artificial genome). As we will see, this approach allows us not only to respect the fundamental definitions of a biological cell, but also to exploit some of the more specialized mechanisms on which the ontogenetic development of a cell is based.

3. Of course, considering the different environments, the analogy will necessarily be limited to the most macroscopic features.

Many of the precise mechanisms whereby the genome is accessed and interpreted inside each cell are still unknown to science[4]. Since our intention is *not* to imitate biology, however, such detailed knowledge is superfluous: the transition from the carbon-based world of biology to the silicon-based world of electronic circuits will necessarily generate considerable discrepancies. Undoubtedly the greatest such discrepancy is a consequence of the process of biological ontogeny itself, which is based on biology's capability to handle physical material: biological ontogenetic development is based on the *physical* growth and replication of cells, and biological self-repair is a consequence of their *physical* creation and destruction. The state of the art in the design of electronic circuits does not allow us to act on the physical structure of our cells (i.e., the silicon) after fabrication.

This obstacle, which was a major factor in preventing the realization of von Neumann's universal constructor, would have been almost insurmountable until a few years ago, when the first FPGA circuits were developed.

2.2.2 Field-Programmable Gate Arrays

An FPGA (Field-Programmable Gate Array) [14, 103] is basically a chip that can be configured (i.e., programmed via software) to realize any given function (that is, to implement any digital logic circuit[5]). They are two-dimensional arrays of logic elements[6] and, while the exact structure of an FPGA element can vary considerably from one manufacturer to the next, certain essential features are constant (Fig. 2-4):

- Each element can implement a programmable function, usually consisting of combinational logic plus one or more memory elements (flip-flops) for sequential behavior. The complexity and the structure of the programmable function can vary considerably from one type of FPGA to the next (for example, the combinational part can be implemented using a lookup table, as in the Xilinx XC4000 [111] and XC5200 [112] families, or be hard-wired, as in the Xilinx XC6200 [113] family).
- Communication between the elements is handled through programmable connections, again of varying complexity depending on the type of FPGA. Experience has shown that connections (rather than functionality) are the main bottleneck for the layout of FPGA circuits, an observation which has led most designers to add long-distance connections distributed inhomogenously throughout the array. As we will see below, this lack of homogeneity is one of the major factors in our decision to design a novel FPGA for the Embryonics project.

4. It is our secret hope that, in some indirect way, our work might be of some use to biologists by suggesting possible mechanisms for accessing and decooding the biological genome.

5. In fact, some FPGAs can implement analog circuits as well.

6. The elements of an FPGA are usually referred to as *cells*, but be will avoid the term so as not to engender confusion with our electronic cells.

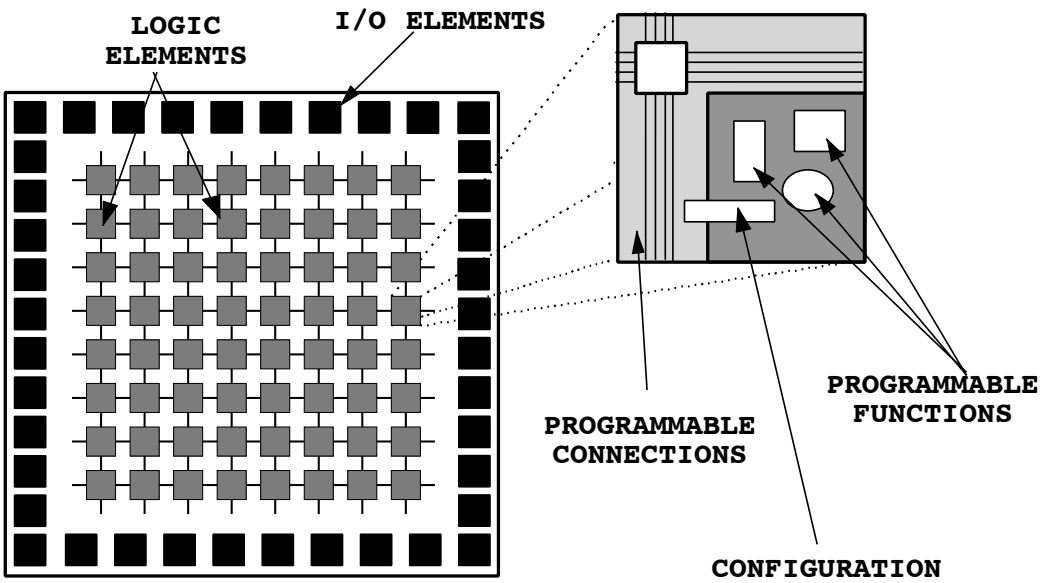

Figure 2-4: Basic structure of a generic FPGA circuit.

- The functionality and the connections of an element are controlled by its configuration. The configuration bitstream (that is, the sum of all the configurations of the FPGA's elements) determines the global behavior of the chip. An FPGA can be configured to implement any digital logic circuit (provided enough elements are available or the circuit can be subdivided among different chips) and in most cases is reprogrammable (that is, its configuration can be erased and replaced by a new one, implementing a different circuit).

FPGAs are quickly becoming an essential tool for the design of complex computer circuits. In the industry, they are mainly used for *rapid prototyping*: manufacturing a VLSI circuit is an expensive and time-consuming endeavor, and a circuit has to be tested very extensively before it is sent to a foundry for fabrication. Most design errors can be detected through software simulation, but the speed (or lack thereof) of such simulations does not allow extensive testing of complex circuits. By using FPGAs to implement a design, the circuit can be tested at hardware speeds, and the debugging can therefore be much faster and more complete (moreover, some faults can only be observed when the circuit is tested at speed).

Obviously, the remarkable versatility of FPGAs comes at a price: speed. The programmable function of each element, being universal (i.e., capable of implementing a number of different functions), is necessarily much slower than a dedicated implementation, and the logic required to make connections programmable inevitably slows the flow of data. As a consequence, circuits implemented using FPGAs can rarely operate at very high clock rates. This limitation, while still allowing FPGAs to abundantly outperform any kind of software simulation for prototyping, is often too restrictive for FPGAs to be used in actual applications.

However, in some cases the versatility of FPGAs can overcome this shortcoming, either because the circuit is not speed-critical, but could benefit from regular upgrades or even dynamic alterations, or because the advantage of having dedicated (often parallel) processors can easily compensate the slower clock rate (for example, specific mathematical operations which would require many clock cycles in a general-purpose processor, but could be executed in a single, if slower, clock cycle in a dedicated processor). Given the prohibitive cost of developing a dedicated computer system in VLSI, using programmable logic can be a viable and cost-effective option in many applications.

Our laboratory has been involved in this kind of FPGA-based systems for many years. Among the most interesting project, we can mention:

- Spyder [42, 43], a VLIW (Very Long Instruction Word) processor with programmable functional units. By dynamically adapting the structure of the functional units, Spyder is able to create what amounts to an instruction-specific processor.
- GENSTORM [68], a dedicated processor for matching DNA or protein sequences with known pattern families. By implementing dedicated algorithms in hardware, GENSTORM can overcome the slower clock rate in a very computationally-intensive application.
- RENCO [15, 34], a reprogrammable network computer (NC): while conventional NCs can charge an application through the network to execute it locally on a general-purpose processor, RENCO can charge the hardware configuration itself, so as to provide a dedicated processor for each application.

Since the Embryonics project is not meant to produce a commercially viable product[7], speed of operation is not one of our main priorities. On the other hand, the reprogrammability of FPGAs is a solution to the problem of implementing an ontogenetic machine, as it provides a way to modify the hardware structure of a system by altering information, sidestepping the need to handle physical matter.

2.2.3 The Artificial Organism

By demonstrating that it is possible to modify hardware using information, the development of FPGA circuits has proved the feasibility of creating computer hardware inspired by biological ontogeny. To develop an approach to the design of such systems, we analyzed the essential features of biological organisms.

- In biology, an organism is a three-dimensional array of cells, all performing their functions in parallel to give rise to global processes (i.e., processes involving the entire organism). Each cell determines its function on the basis of its position within the array (cellular differentiation), during both growth and healing. To respect the biological analogy, our electronic organism will then consist of a two-

7. At least, not directly: we *do* hope that some of the strategies and mechanism developed in our project will eventually, duly adapted, have repercussions on mainstream circuit design.

dimensional array of elements[8] working in parallel to achieve a global task, i.e. a given application.

- In biology, each cell contains the entire genome, that is, the function of every cell in the organism. If we are to maintain the analogy between the genome and a computer program, this feature implies that we must regard the elements of our electronic organism as processors, each containing the same program. No single cell uses the entire genome, accessing only those portions necessary to perform its functions. Similarly, no single processor will execute all the instructions in its program, but will use its position within the array to identify which subset of the program to access[9].
- In biology, not all cells have access to external stimuli. Input from the outside world is limited to a (relatively) small subset of all the cells, and is propagated throughout the organism via messages passed from one cell to another. We can make use of this concept to determine a communication network for our electronic organism: since we do not need all processors to have access to external inputs and outputs, we can limit our network to local connections (for example, a processor's connections can be limited to its cardinal neighbors), thus greatly simplifying the communication network[10].

Drawing inspiration from biological organisms has thus led to define our organism as a two-dimensional array of processing elements, all identical in structure (each cell must be able to execute any subset of the genome program) and each executing a different part of the same program, depending on its position. At first glance, this kind of system might not seem very efficient from the standpoint of conventional circuit design: storing a copy of the genome program in each processor is redundant, since each processor will only execute a subset.

However, by accepting the weaknesses of bio-inspiration, we can also partake of its strengths. One of the most interesting features of biological organisms is their robustness, a consequence of the same redundancy which we find wasteful: since each cell stores a copy of the entire genome, it can theoretically replace any other. Thus, if one or more cells should die as a consequence of a trauma (such as a wound), they can be recreated starting from any other cell. By analogy, if one or more of our processors should “die” (as a consequence, for example, of an hardware fault), they can theoretically be replaced by any other processor in the array.

8. When dealing with the world of electronics circuits, the current state of the art in circuit fabrication limits us to two dimensions, but this limitation (being quantitative, rather than qualitative) is not particularly constraining, and should not have serious consequences for our project.

9. This kind of parallel systems are a particular case of what are commonly known as Single-Program Multiple-Data (SPMD) computer systems [5].

10. Complex organisms can have rather complicated mechanisms which allow long-distance communication between cells to occur, but these can be realized using local connections, with a loss of efficiency compensated by the gain in simplicity.

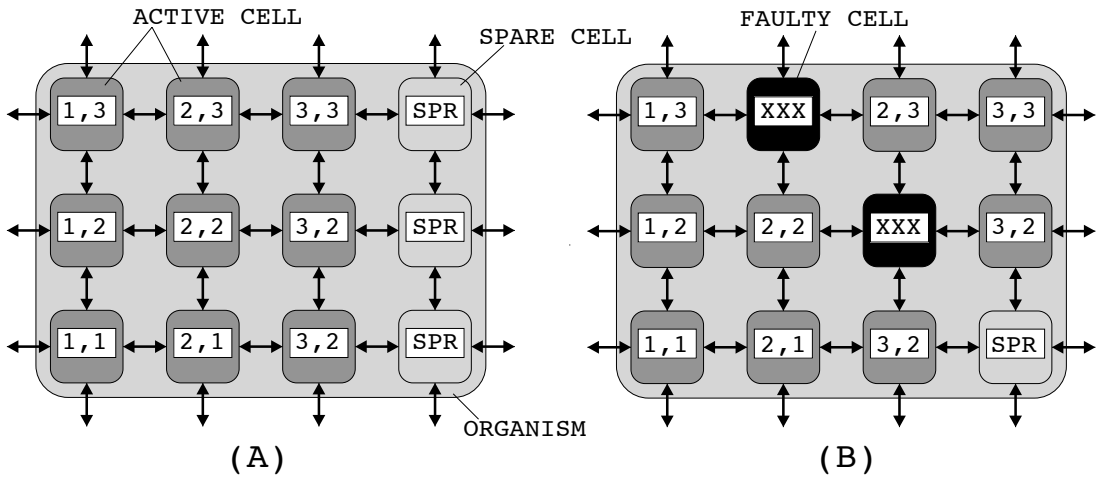

Figure 2-5: A small (4x3 cells) artificial organism with no faulty elements (A) and after reconfiguration (B).

The redundancy introduced by having multiple copies of the same program thus provides an intrinsic support for self-repair, one of the main objectives of our research: by providing a set of spare cells (i.e., cells that are inactive during normal operation, but which are identical to all other cells and contain the same genome program), we are able (Fig. 2-5) to reconfigure the array around one or more faulty processors (of course, as in living beings, too many dead cells will result in the death of the entire organism).

Moreover, if the function of a cell depends on its coordinates, the task of self-replication is greatly simplified: by allowing our coordinates to cycle (Fig. 2-6) we can obtain multiple copies of an organism with a single copy of the program (provided, of course, that enough processors are available). Depending on the application and on the requirements of the user, this feature can be useful either by providing increased performance (multiple organisms processing different data in parallel) or by introducing an additional level of robustness (the outputs of multiple organisms processing the same data can be compared to detect errors).

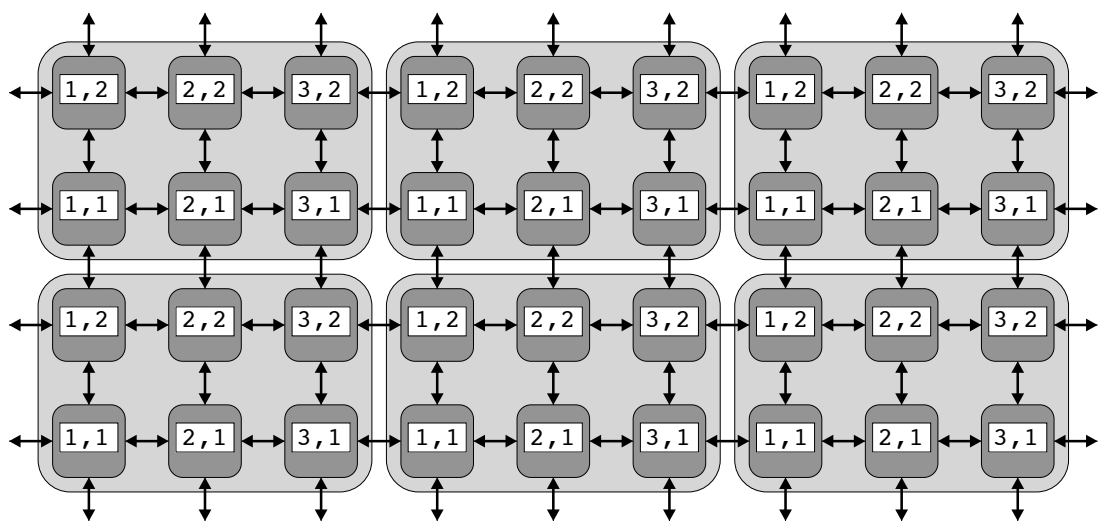

Figure 2-6: Multiple copies of the organism through coordinate cycling.

2.2.4 The Artificial Cell

Keeping in mind the requirements of the organism, we can now determine the basic features of our electronic cell. At the hardware level, all cells must be identical: since we want our organism to be reprogrammable, we cannot fix *a priori* the functionality of our cell. In addition, it has to be able to store the genome program with a coordinate-dependent access mechanism. The minimal features of our cells must therefore include (Fig. 2-7):

- A memory to store the genome. The size of the genome is application-dependent, and therefore we should ideally be able to configure the size of the memory.
- An `[X,Y]` coordinate system, to allow the cell to locate its position within the array, and thus its function (Fig. 2-8). The size of the registers storing the coordinates limits the maximum size of the array. Since there should be no limit to the size of an organism, the size of the coordinate registers should be programmable.
- An interpreter to read and execute the genome. The complexity of this interpreter can vary depending on the application, but once a language for the genome program has been defined (see below), we should be able to keep it more or less constant.
- A functional unit, for data processing. It can contain a variety of logic elements, from a single register to a full arithmetic unit and beyond, depending on the application. While it might be more efficient to tailor the functional unit to the application, we can however execute any program with a fixed functional unit by shifting the complexity to the genome program.
- A set of connections handled by a routing unit. In theory, cardinal connections (where each cell communicates with its neighbors in the four cardinal directions) are sufficient for any application. However, more complex, application-dependent patterns can be considered.

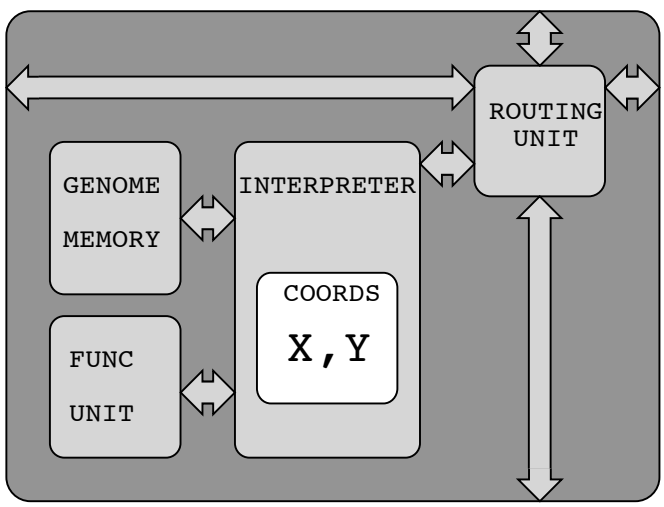

Figure 2-7: Structure of an artificial cell.

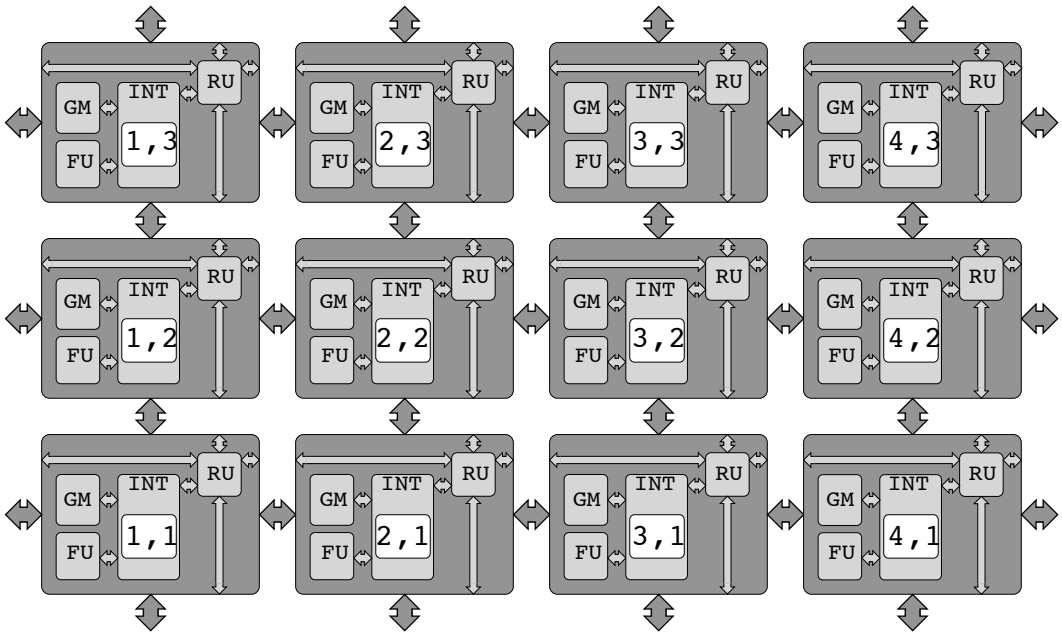

Figure 2-8: A small array of artificial cells.

While the structure of the functional unit can vary, biological organisms provide a powerful example of complex behavior derived not from the complexity of the components, but from the parallel operation of many simple elements. One of the goals of our project is to show that biological inspiration allows us, without excessive difficulty, to design complex systems by combining very simple cells.

To illustrate the capabilities of our system, we have realized in hardware a first prototype of such a cell, known as *MicTree* (for *tree of microinstructions*) [58, 59, 60, 93], where the functional unit is a 4-bit register, the coordinate registers are 4 bits wide (limiting the maximum size of organism to 16x16 cells, sufficient for demonstration purposes, but restrictive for real applications), and communications, implemented using directional 4-bit busses, are limited to the cardinal neighbors. To write genome programs for MicTree, we have developed a small (but complete) language, complex enough to potentially implement any application, but simple enough that the interpreter need not be large. The instructions of the language are as follows:

- **if** VAR **else** LABEL
- **goto** LABEL
- **do** REG **=** DATA [**on** MASK]
- **do** X **=** DATA
- **do** Y **=** DATA
- **do** VAROUT **=** VARIN
- **nop**

where REG is the 4-bit register in the functional unit, X and Y are the two 4-bit coordinates, MASK lets the bits of REG be accessed individually, and VAROUT and VARIN identify one of the four possible I/O busses. The instructions are coded in 8-bit words, and the genome memory, of fixed size, can store up to 1024 such words.

Each MicTree cell, realized using an Actel A1020B [3] FPGA (ideally suited for prototyping), was then mounted on a custom printed circuit board with a set of 7-segment displays and LEDs (Light Emitting Diodes), which in turn was inserted in an 8X8cm plastic box, the Biodule 601 (Fig. 2-9). These boxes can be fitted together, like a jigsaw puzzle, to form a two-dimensional array of cells and provide a set of neighbor-to-neighbor connections without additional cables.

In order to demonstrate the features of our cellular system, we used our prototype to implement a set of applications:

- the *BioWatch* [60], a timer which keeps count of minutes and seconds, realized with four MicTree cells;
- a *random number generator* [59, 60], a one-dimensional non-uniform cellular automaton, realized with five MicTree cells;
- a *specialized Turing machine* [58, 60, 93], implementing a *parenthesis checker* (that is, a program which decides whether a sequence of parentheses is well-formed), realized using ten MicTree cells.

These applications, because of the limited size of our prototype, are relatively simple. Nevertheless, they are interesting in that they exhibit both the properties of self-repair (provided spare cells are available) and self-replication (if the array is large enough to contain multiple copies of the organism). In designing applications for our prototype, however, we came to realize that, while MicTree contains all of the minimal features required by our electronic cell, its fixed structure seriously limits the range of applications it can efficiently implement.

One of the most constraining factors is probably the functional unit: a single 4-bit register is too small for complex applications. Of course, the functional unit is not, by itself, constraining, as it could be compensated by using either a larger organism (cells can be very simple, if many are working in parallel) or a longer

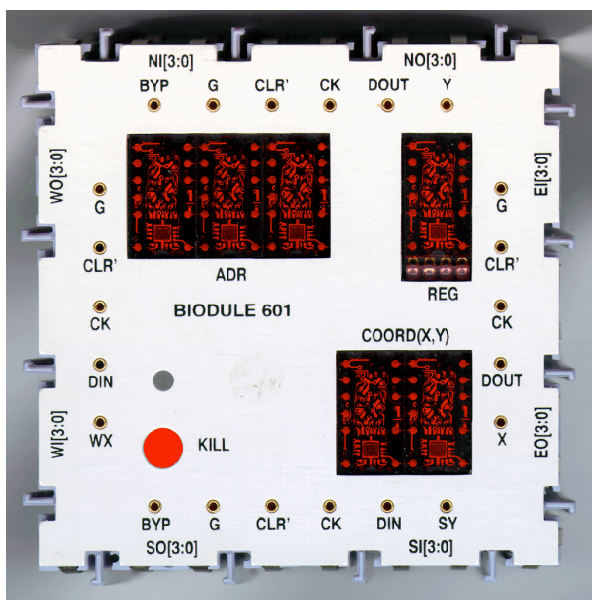

Figure 2-9: Prototype of an artificial cell: the Biodule 601.[Photo by André Badertscher]

genome program. However, since both the scope of the coordinates and the size of the genome memory are also fixed, MicTree is unsuited for complex applications.

Therefore, as we suspected, we need to able to reprogram the hardware itself to create a truly ontogenetic machine. The features of the functional unit (e.g., the size of the data path) have to fit the application to reduce the complexity on the rest of the cell. The genome can have very different sizes depending on the complexity of the task (there usually is a trade-off between hardware and software, that is between the complexity of the functional unit and the length of the genome), and the size of the genome memory should therefore be programmable, a feature has an impact on the complexity of the interpreter. The size of the registers containing the coordinates depends on the size of the organism. A particular application might be more efficiently executed by a specific connection pattern.

Obviously, we can design a complex enough cell that it will satisfy the greatest majority of possible applications. In fact, with a complex enough interpreter (and thus genome language) or functional part, we can be limited only by size of the genome memory. In practice, however, this solution would be very wasteful: no benefit could be achieved through tailoring processor to application, and the processors would become so complex that we would reintroduce all the drawbacks which have kept conventional multiprocessor parallel systems from gaining widespread acceptance (such as the difficulty of writing parallel code and the complexity of handling communication between processors).

Once again, biology provides a possible solution: the physical structure of a biological cell is determined by chemical processes occurring at the *molecular level*. Having introduced the concepts of electronic organism and of electronic cell, we now need to define the concept of electronic molecule. Fortunately, we are familiar with a kind of circuit which can be used to implement our molecular layer: FPGAs. Using programmable gate arrays as our molecules allows us to maintain the fundamental analogy with the world of biology: whereas a living cell consists of a three-dimensional array of molecules, a processor consists of a two-dimensional array of logic gates. Further, since in biology the most complex mechanisms of the cellular layer (notably, the genome program and the coordinate system) do not concern the molecular layer, the structure of our electronic molecule is not unlike that of conventional FPGA logic elements. We are therefore confronted with a three-layer system (Fig. 2-10), summarized in Table 2-1.

Biology	**Electronics**
multi-cellular Organism	Parallel computer system
Cell	Processor
Molecule	FPGA element

Table 2-1: Analogies between biological and electronic systems in Embryonics.

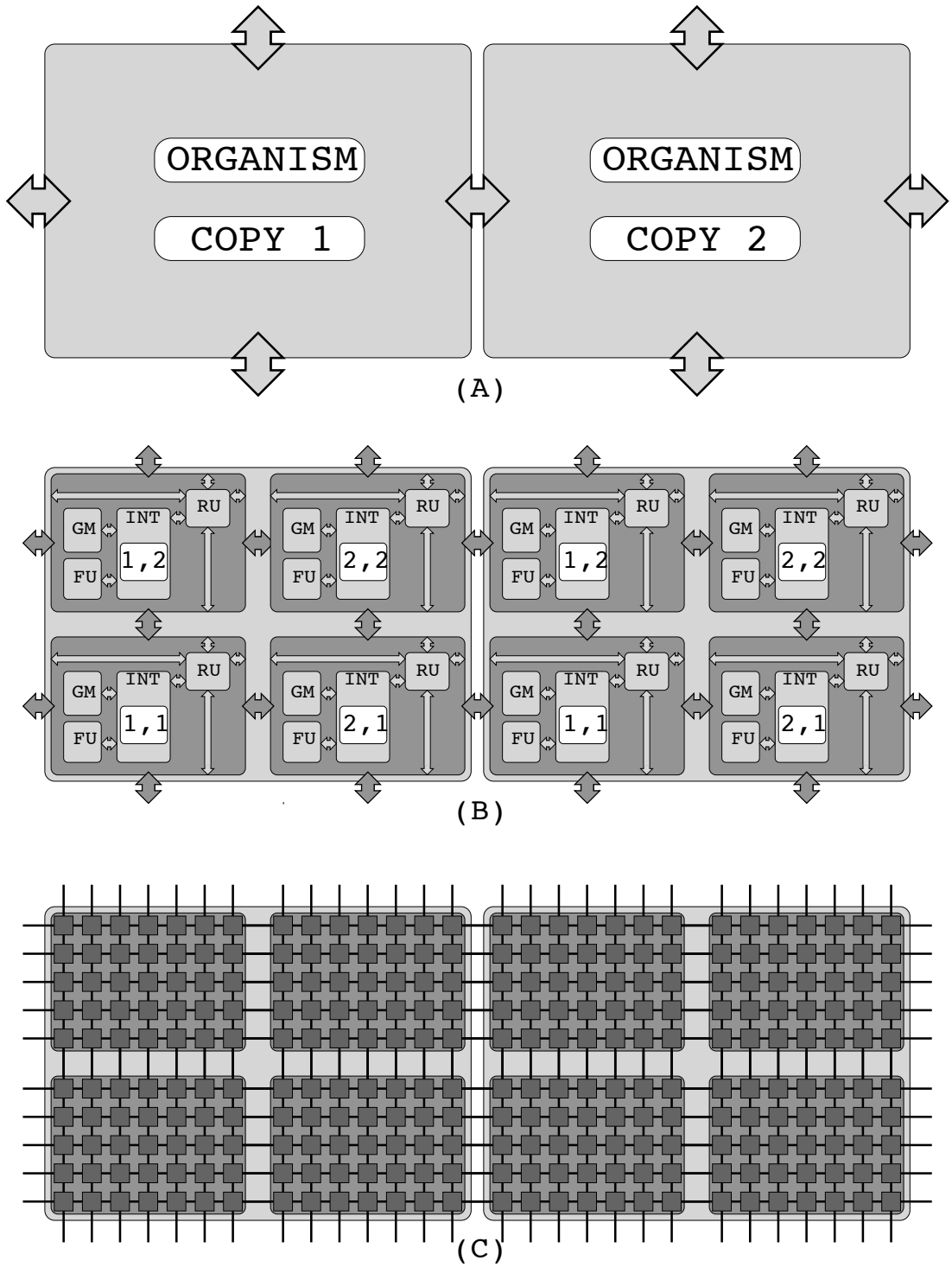

Figure 2-10: The three-level ontogenetic hardware: (A) organism (system) level, (B) cellular (processor) level, and (C) molecular (FPGA) level.

2.2.5 The Artificial Molecule

As we have seen, FPGAs are necessary for a hardware realization of our electronic cell, and they thus assume a fundamental importance for the Embryonics project. In theory, any commercially available FPGA could be used to implement one of our cells. In practice, however, the fundamental features of ontogenetic hardware can not easily implemented using conventional programmable circuits.

The first such feature concerns the basic structure of our artificial organism: a two-dimensional array of identical processing elements (cells). The dimensions of such an array can be very substantial, since a processor, however simple, is nevertheless a relatively complex circuit, and restricting the amount of programmable logic to a single chip can quickly become a serious constraint. Envisaging the use of more than one chip, however, implies the presence of a mechanism allowing designs to be easily distributed among multiple chips. Three main factors inhibit this process in conventional FPGAs:

- No currently available routing software (the program which generates a configuration bitstream from the layout of a circuit) is efficient in exploiting the regularity of the array, i.e., the fact that our organism is an array of identical processing elements. This problem could be alleviated by the use of *hard macros* (i.e., hand-placing the functions and connections on the elements of the FPGA), an extremely time-consuming process which would have to be repeated whenever the structure of the cell changed (i.e., for every new application), and thus is not a practical option. Hopefully, future software will be able to recognize and exploit such regularity, but it is doubtful whether such a feature represents a high priority for FPGA designers.
- Partitioning software (the program which divides a large design into sub-circuits to exploit multiple chips) is still at an early stage of development, and is not very efficient. As FPGAs become more and more used for prototyping complex systems, ameliorating the performance of such programs is a relatively high-priority task for developers, but it is proving difficult, mainly because of the non-homogeneous structure of commercial FPGAs.
- Commercial FPGAs are not homogeneous. All currently available FPGAs have, at best, a semi-homogenous structure: the need for long-distance connections (indispensable for many kinds of circuits, but not a requirement for systolic arrays) introduces inhomogeneities in the array. This feature, so useful for many applications, is a rather serious inconvenient for our organism, where the size of the cell is application-dependent: the presence of inhomogeneities in the structure of the array imposes restrictions on the size of the cell.

The second feature of ontogenetic hardware we need to implement is healing, i.e., in a terminology more familiar to computer designers, self-repair.

As we mentioned, the cellular level, through its coordinate system, provides an intrinsic capability for self-repair. However, while in biological systems the death of a cell is a commonplace occurrence, in our electronic organism we can not afford the death of many cells: if we assume that a cell will require something in the order of a few hundred molecules, and we assume a finite number of available molecules, we obviously need to minimize the loss of cells.

As a consequence, we need a mechanism to repair minor faults at the molecular level, which in turn requires a mechanism which allows self-test, i.e. the detection of faults. In fact, a prerequisite for being able to repair faults, both at the cellular and at the molecular level, is the capability of detecting the occurrence of such faults. Moreover, if the self-repair mechanism is to operate on-line (i.e., while the application is running, rather than in a separate phase), self-test must also be performed on-line.

The design of such a mechanism is a considerable challenge, particularly since the size of the FPGAs elements is such that the additional logic required by the process be minimized. However, the absence of molecular-level self-test would imply that the detection of faults be handled at the cellular level, and since the cells are application-dependent, the self-test logic would have to be included in every new cell, which would complicate their design to a considerable degree.

Providing an on-line self-test and self-repair mechanism at the molecular level is therefore, if not required, at least very desirable, since it would not only increase the robustness of the system (which would "survive" a larger number of faults), but also substantially simplify the design of the cells (by moving much of the additional logic to the application-independent molecular level). Fortunately, the task of designing such a mechanism is somewhat simplified by the presence of a second level of robustness at the cellular level, which implies that the molecular-level self-repair need not be able to handle all faults occurring in the circuit.

To efficiently implement our electronic organism, we would therefore require a completely homogenous self-repairing FPGA which can easily be configured as an array of identical processing elements (self-replication). No commercial FPGA provides the features we need, and it is doubtful whether the demand for self-repair and self-replication is sufficient to justify the additional logic in a commercial circuit, at least in the foreseeable future. As consequence, the efficient implementation of ontogenetic hardware requires the conception of a new FPGA, capable of self-replication and self-repair. The development of such an FPGA constitutes the main challenge of the ontogenetic axis of the Embryonics project, and is the focus of this thesis.

CHAPTER 3

SELF-REPLICATION

The conception of a self-repairing and self-replicating FPGA presents a considerable challenge. If self-repair is a relatively well-investigated feature in the design of digital logic circuits, which implies the existence of a certain number of approaches to the implementation of self-repairing systems and thus an existing base for the development of our system, the same cannot be said for the property of self-replication: research in this domain is relatively scarce, particularly where hardware is concerned. This lack compelled us to develop an entirely new self-replication mechanism, allowing us to arrive at an efficient hardware realization.

We will begin this chapter with a short introduction to *cellular automata*, the computational model traditionally used in the study of self-replicating computing machines. We will then analyze existing self-replicating automata, and notably von Neumann's *universal constructor* and Langton's *loop*, before discussing the automata we developed in Embryonics. In the last section we will then describe the process we followed to adapt our self-replication mechanism to FPGAs.

3.1 Cellular Automata

The first phase of this development of our self-replicating system was, naturally, to examine the existing approaches in order to identify their strengths and weaknesses in relation to our requirements. Most, if not all, of the relevant research was applied to *cellular automata* (CA) [20, 60, 101, 109], a computational model which, while both ill-suited for hardware implementations and difficult to manipulate, nevertheless provides a relatively strict mathematical framework for the development of self-replicating machines, and has traditionally been the environment of choice for the study of such systems.

Cellular automata are arrays of elements, or *cells*[1], all behaving identically[2], depending on the element's *state*. At regular, discrete intervals (*iterations*), the state of all elements is updated, depending on the current state of the element itself and that of its neighbors, according to a set of *transition rules*.

1. Again, standard terminology creates a problem with regards to our project. The elements of cellular automata are usually referred to as cells, but do not really resemble biological cells (they do not contain a genome). We will thus call them *elements* or sometimes *molecules*, the closest biological analog.
2. Elements with different behaviors are allowed in some CAs, usually called *non-uniform*. Since all the models used in this thesis are uniform, however, we can consider all elements to have identical behavior.

In order to illustrate the operation of cellular automata, we can examine one of the best known (and simplest) two-dimensional[3] CA, commonly referred to as *Life* [32]. Life is a uniform two-dimensional cellular automaton where each element can be in one of only two states (*alive* or *dead*). The next state of an element depends on its current state and that of its eight closest neighbors (to the north, south, east, west, northeast, southeast, southwest, and northwest), and is calculated from a set of simple rules:

- if fewer than two elements in the neighborhood are alive, the next state is dead (death by starvation);
- if more than three elements in the neighborhood are alive, the next state is dead (death by overcrowding);
- if exactly three elements in the neighborhood are alive, then the next state is alive (birth);
- otherwise (i.e., if exactly two of the elements in the neighborhood are alive) the next state is equal to the current state (survival).

This very simple automaton is obviously not very powerful: the majority of initial configurations (the ensemble of the states of all elements at iteration 0) lead either to an empty space or to a collection of small, isolated cyclic patterns. One of the best-known of these patterns is the *glider* (Fig. 3-1), a small structure capable of moving diagonally across the space. The glider is one of building blocks used to create machines which can become extremely complex (Fig. 3-2).

To resume, a cellular automaton is defined by the following parameters:

- A number of dimensions, usually one or two, rarely three, and almost never four or more. All the automata used to model self-replication are two-dimensional, as is Life.
- A set of states (two in the case of Life) and an initial configuration, defining the state of all the elements of the array at iteration 0. While there is no theoretical limit to the number of states in an automaton, for practical considerations very few automata use more than a handful.

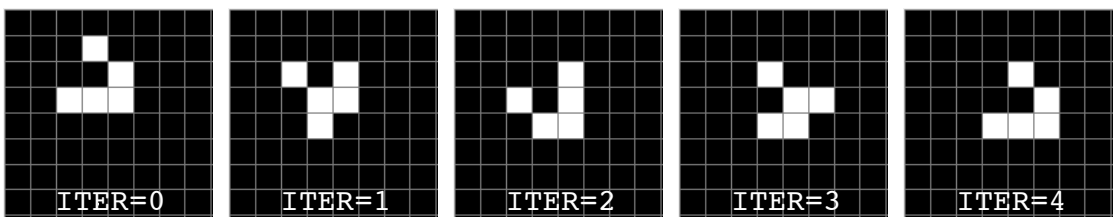

Figure 3-1: The glider, a simple cyclic structure capable of moving through space in the Life cellular automaton.

3. There is no theoretical limit to the number of dimensions of a cellular automaton. However, CA being, essentially, graphic-oriented models, the need to display their activity imposes a practical limit: the greatest majority of existing automata are either one- or two-dimensional, and while we are aware of the existence of some three-dimensional automata, we have no knowledge of any implementation in four or more dimensions. All of the automata used to study self-replication are two-dimensional.

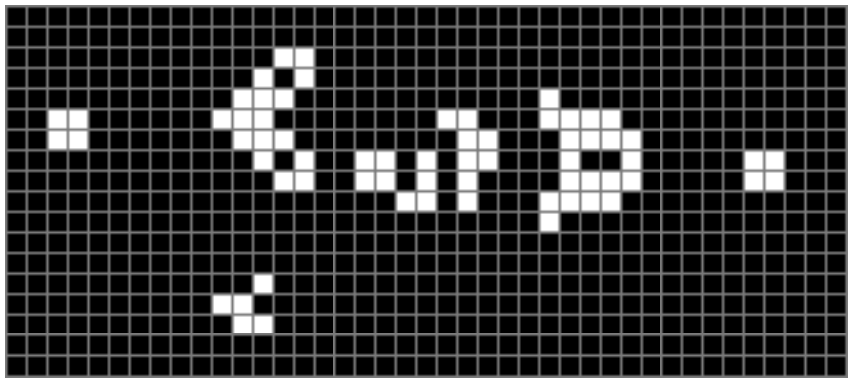

Figure 3-2: The glider gun, a moderately complex structure capable of generating gliders ar regular intervals.

- A neighborhood, which specifies which neighbors will have an effect on an element's next state. By far the most common for two-dimensional automata are the neighborhood of 5 (the element itself plus its cardinal neighbors to the north, south, east, and west) and that of 9 (the element itself plus its neighbors to the north, south, east, west, northeast, southeast, southwest, and northwest). Life uses a neighborhood of 9.
- A collection of transition rules, used to compute an element's next state depending on the neighborhood. The rules can be expressed either as an algorithm (as for Life above) or exhaustively as a lookup table[4]. In the latter case, the total number of rules necessarily to exhaustively define a cellular automaton is S^N, where S is the number of states and N the neighborhood (thus, to completely define Life, a lookup table of 2^9=512 rules would be required). In practice, the number of required rules can, in many cases, be considerably reduced[5], but the lookup table for a complex automaton (i.e., one with many states) can nevertheless reach a very important size.

As should now be obvious, cellular automata are not a model which can easily be applied to digital hardware. The need for each element to access the transition rules, coupled with the large number of elements required for complex behavior, is a serious drawback for an electronic implementation. In addition, while cellular automata can be a powerful model for certain kinds of applications where the transition rules are relatively simple and known in advance (e.g., the modelization of the behavior of gases), the design of a complex automaton is extremely difficult: the absence of global rules imply that complex automata (i.e., configurations which span a large number of elements) have to be handled exclusively at the local level, and no formal approach is available to aid in this task.

4. A typical lookup table entry for a 2-state 9-neighbors automaton would have the form:

```
0,0,0,0,1,0,1,1,0=1;
```

where the first nine values represent the current state of the neighborhood and the last is the next state.

5. For example by assuming a "default" behavior (e.g., the next state will remain equal to the current state unless a rule specifies otherwise).

3.2 Von Neumann's Universal Constructor

The self-replication of digital logic circuits has thus far engendered a relatively modest amount of research, probably, as we mentioned, because of the technological problems associated with the implementation of such a feature. The existing approaches to the self-replication of computing systems are essentially derived from the work of John von Neumann [10], who pioneered this field of research. Unfortunately, the state of the art in the fifties restricted von Neumann's investigations to a purely theoretical level, and the work of his successors mirrored this constraint. In this section, we will analyze von Neumann's research on the subject of self-replicating computing machines, and in particular his *universal constructor*, a self-replicating cellular automaton [104].

3.2.1 Von Neumann's Self-Replicating Machines

Von Neumann, confronted with the lack of reliability of computing systems[6], turned to nature to find inspiration in the design of fault-tolerant computing machines. Natural systems are among the most reliable complex systems known to man, and their reliability is a consequence not of any particular robustness of the individual cells (or organisms), but rather of their extreme redundancy. The basic natural mechanism which provides such reliability is self-reproduction[7], both at the cellular level (where the survival of a single organism is concerned) and at the organism level (where the survival of the species is concerned).

Thus von Neumann, drawing inspiration from natural systems, attempted to develop an approach to the realization of self-replicating computing machines (which he called *artificial automata*, as opposed to *natural automata*, that is, biological organisms). In order to achieve his goal, he imagined a series of five distinct models for self-reproduction [104, p. 91-99]: the *kinematic* model, the *cellular* model, the *excitation-threshold-fatigue* model, the *continuous* model, and the *probabilistic* model.

- The kinematic model, introduced by von Neumann on the occasion of a series of five lectures given at the University of Illinois in December 1949, is the most general. It involves structural elements such as sensors, muscle-like components, joining and cutting tools, along with logic (switch) and memory elements. Concerning, as it does, physical as well as electronic components, its goal was to

6. Let us remember that the computers von Neumann was familiar with were based on vacuum-tube technology, and that vacuum tubes were much more prone to failure than modern transistors. Moreover, since the writing and the execution of complex programs on such systems represented many hours (if not many days) of work, the failure of a system had important consequences in wasted time and effort.

7. You will note that we use the terms *self-replication* and *self-reproduction* interchangeably. In reality, the two terms are not really synonyms: self-reproduction is more properly applied to the reproduction of organisms, while self-replication concerns the cellular level. As we will see, the more correct term to use in this circumstance would probably be self-replication, but since von Neumann favored self-reproduction, we will ignore the distinction.

define the bases of self-replication, but was not designed to be implemented.

- In order to find an approach to self-replication more amenable to a rigorous mathematical treatment, von Neumann, following the suggestion of the mathematician S. Ulam, developed a cellular model. This model, based on the use of cellular automata as a framework for study, was probably the closest to an actual realization. Even if it was never completed, it was further refined by von Neumann's successors and was the basis for all further research on self-replication.
- The excitation-threshold-fatigue model was based on the cellular model, but each cell of the cellular automaton was replaced by a neuron-like element. Von Neumann never defined the details of the neuron, but through a careful analysis of his work, we can deduce that it would have borne a fairly close relationship to today's artificial neural networks, with the addition of some features which would have both increased the resemblance to biological neurons and introduced the possibility of self-replication.
- For the continuous model, von Neumann planned to use differential equations to describe the process of self-reproduction. Again, we are not aware of the details of this model, but we can assume that von Neumann planned to define systems of differential equations to describe the excitation, threshold and fatigue properties of a neuron. At the implementation level, this would probably correspond to a transition from purely digital to analog circuits.
- The probabilistic model is the least well-defined of all the models. We know that von Neumann intended to introduce a kind of automaton where the transitions between states were probabilistic rather than deterministic. Such an approach would allow the introduction of mechanisms such as mutation and thus of the phenomenon of evolution in artificial automata. Once again, we cannot be sure of how von Neumann would have realized such systems, but we can assume they would have exploited some of the same tools used today by genetic algorithms.

Of all these models, the only one von Neumann developed in some detail was the cellular model. Since it was the basis for the work of his successors, it deserves to be examined more closely.

3.2.2 Von Neumann's Cellular Model

In von Neumann's work, self-reproduction is always presented as a special case of *universal construction*, that is, the capability of building any machine given its description (Fig. 3-3). This approach was maintained in the design of his cellular automaton, which it therefore much more than a self-replicating

machine. The complexity of its purpose is reflected in the complexity of its structure, based on three separate components:

- A memory tape, containing the description of the machine to be built, in the form of a uni-dimensional string of elements. In the special case of self-reproduction, the memory contains a description of the universal constructor itself[8](Fig. 3-4).
- The constructor itself, a very complex machine capable of reading the memory tape and interpreting its contents.
- A constructing arm, directed by the constructor, used to build the offspring (i.e., the machine described in the memory tape). The arm moves across the space and sets the state of the elements of the offspring to the appropriate value.

The implementation as a cellular automaton is no less complex. Each element has 29 possible states, and thus, since the next state of an element depends on its current state and that of its four cardinal neighbors, 29^5=20,511,149 transition rules are required to exhaustively define its behavior. If we consider that the size of von Neumann's constructor is of the order of 100,000 elements, we can easily understand why a hardware realization of such a machine is not really feasible.

In fact, as part of the Embryonics project, we did realize a hardware implementation of a set of elements of von Neumann's automaton [12, 89]. By carefully designing the hardware structure of each element, we were able to considerably reduce the amount of memory required to host the transition rules. Nevertheless, our system remains a demonstration unit, as it consists of a few elements only, barely enough to illustrate the behavior of a tiny subset of the entire machine.

Before we continue, we should mention that von Neumann went one step further in the design of his universal constructor. If we consider the universal constructor from a biological viewpoint, we can associate the memory tape with the genome, and thus the entire constructor with a single cell (which would imply a parallel between the automaton's elements and molecules).

However, the constructor, as we have described it so far, has no functionality outside of self-reproduction. Von Neumann recognized that a self-replicating machine would require some sort of functionality to be interesting from an engineering point of view, and postulated the presence of a *universal computer* (in practice, a universal Turing machine, an automaton capable of performing any computation) alongside the universal constructor (Fig. 3-5).

Von Neumann's constructor can thus be regarded as a *unicellular* organism, containing a genome stored in the form of a memory tape, read and interpreted by the universal constructor (the mother cell) both to determine its operation and to direct the construction of a complete copy of itself (the daughter cell).

8. The memory of von Neumann's automaton bears a strong resemblance to the biological genome. This resemblance is even more remarkable when considering that the structure of the genome was not discovered until after the death of von Neumann.

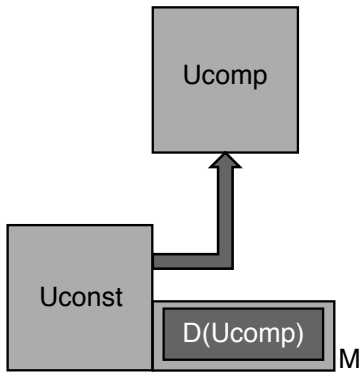

Figure 3-3: Von Neumann's universal constructor *Uconst* can build a specimen of any machine (e.g., a universal Turing machine *Ucomp*) given its description *D(Ucomp)*.

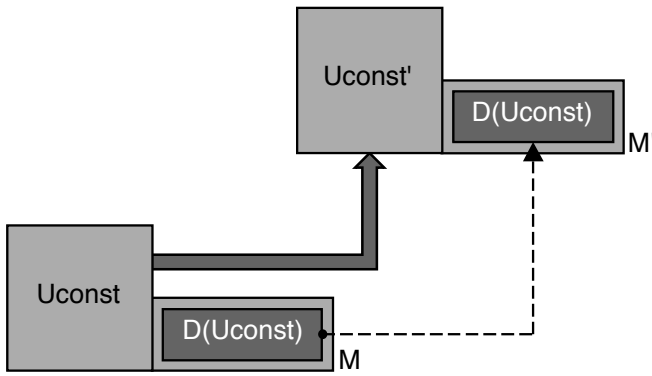

Figure 3-4: Given its own description *D(Uconst)*, von Neumann's universal constructor is capable of self-replication.

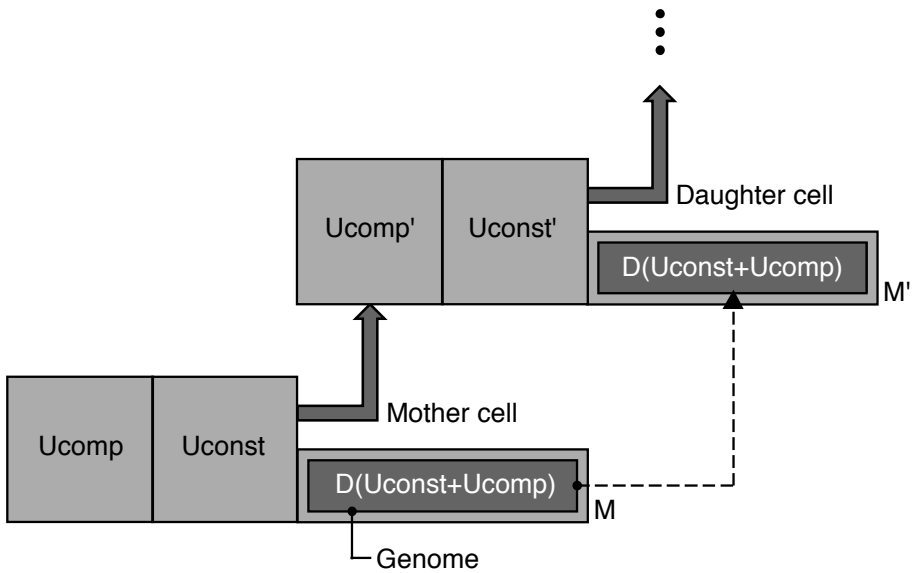

Figure 3-5: By extension, Von Neumann's universal constructor can include a universal computer and still be capable of self-replication.

3.2.3 Von Neumann's Successors

The extreme size of von Neumann's universal constructor has so far prevented any kind of physical implementation (apart from the small demonstration unit we mentioned). But further, even the simulation of a cellular automaton of such complexity was far beyond the capability of early computer systems. Today, such a simulation is starting to be conceivable. Umberto Pesavento, a young Italian high school student, developed a simulation of von Neumann's entire universal constructor [78]. The computing power available did not allow him to simulate either the entire self-replication process (the length of the memory tape needed to describe the automaton would have required too large an array) or the Turing machine necessary to implement the universal computer, but he was able to demonstrate the full functionality of the constructor. Considering the rapid advances in computing power of modern computer systems, we can assume that a complete simulation could be envisaged within a few years.

The impossibility of achieving a physical realization did not however deter some researchers from trying to continue and improve von Neumann's work [11, 54, 71]. Arthur Burks, for example, in addition to editing von Neumann's work on self-replication [17, 104], also made several corrections and advances in the implementation of the cellular model. Codd [20], by altering the states and the transition rules, managed to simplify the constructor by a considerable degree. However, without in any way lessening these contributions, we can say that no major theoretical advance in the research on self-reproducing automata occurred until C. Langton, in 1984, opened a second stage in this field of research.

3.3 Langton's Loop

Von Neumann's Universal Constructor was so complex because it tried to implement self-reproduction as a particular case of construction universality, i.e. the capability of constructing any other automaton, given its description. C. Langton approached the problem somewhat differently, by attempting to define the simplest cellular automaton capable exclusively of self-reproduction.

As a consequence of this approach, his automaton, commonly known as *Langton's Loop* [53], is orders of magnitude simpler than von Neumann's. In fact, it is loosely based on one of the simplest organs[9] in Codd's automaton: the periodic emitter (itself derived from von Neumann's periodic pulser), a relatively simple structure capable of generating a repeating string of a given sequence of pulses.

Langton's loop (Fig. 3-6) is named after the dynamic storage of data inside a square sheath (red in the figure). The data is stored as a sequence of instructions for directing the constructing arm, coded in the form of a set of three states. The data turns counterclockwise in permanence within the sheath, creating a loop.

9. An organ in this context can be seen as a self-supporting structure capable of a single sub-task.

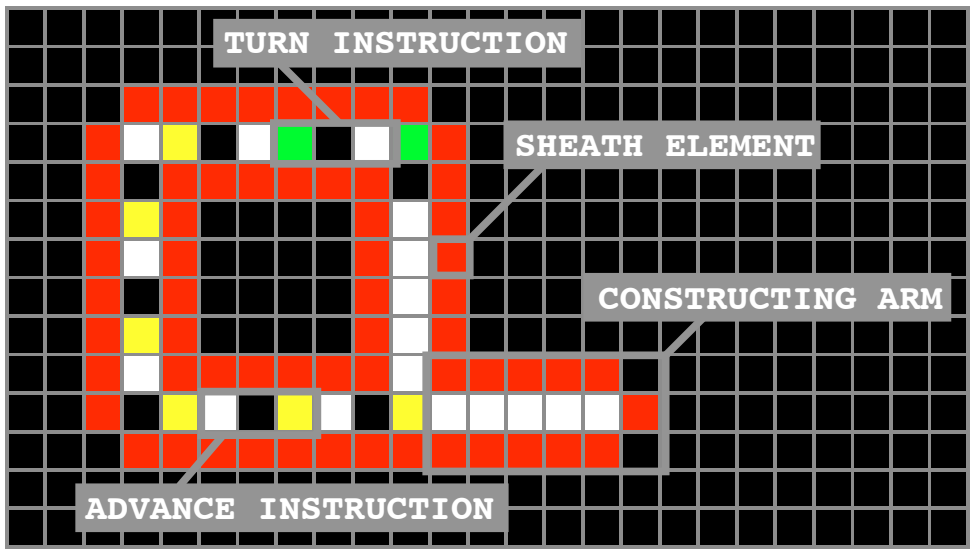

Figure 3-6: The initial configuration of Langton's Loop (iteration 0).

The two instructions in Langton's loop are extremely simple. One instruction (uniquely identified by the yellow element in the figure) tells the arm to advance by one position (Fig. 3-7), while the other (green in the figure) directs the arm to turn 90° to the left (Fig. 3-8). Obviously, after three such turns, the arm has looped back on itself (Fig. 3-9), at which stage a messenger (the pink element) starts the process of severing the connection between the parent and the offspring, thus concluding the replication process. Once the copy is over, the parent loop proceeds to construct a second copy of itself in a different direction (to the north in this example), while the offspring itself starts to reproduce (to the east in this example).

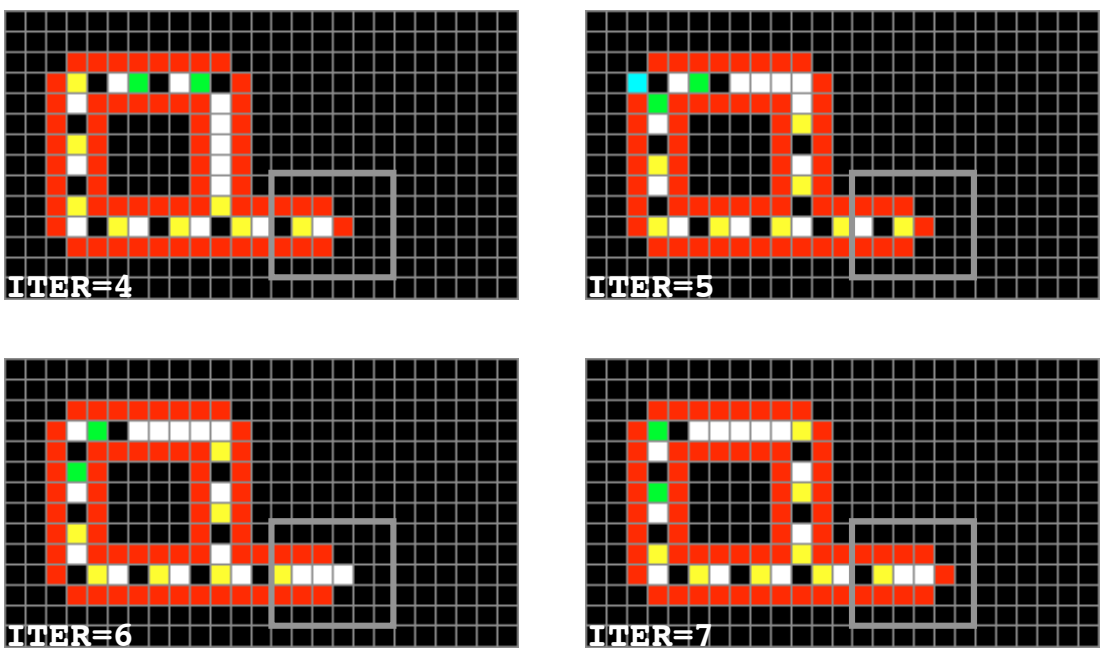

Figure 3-7: The constructing arm advances by one space.

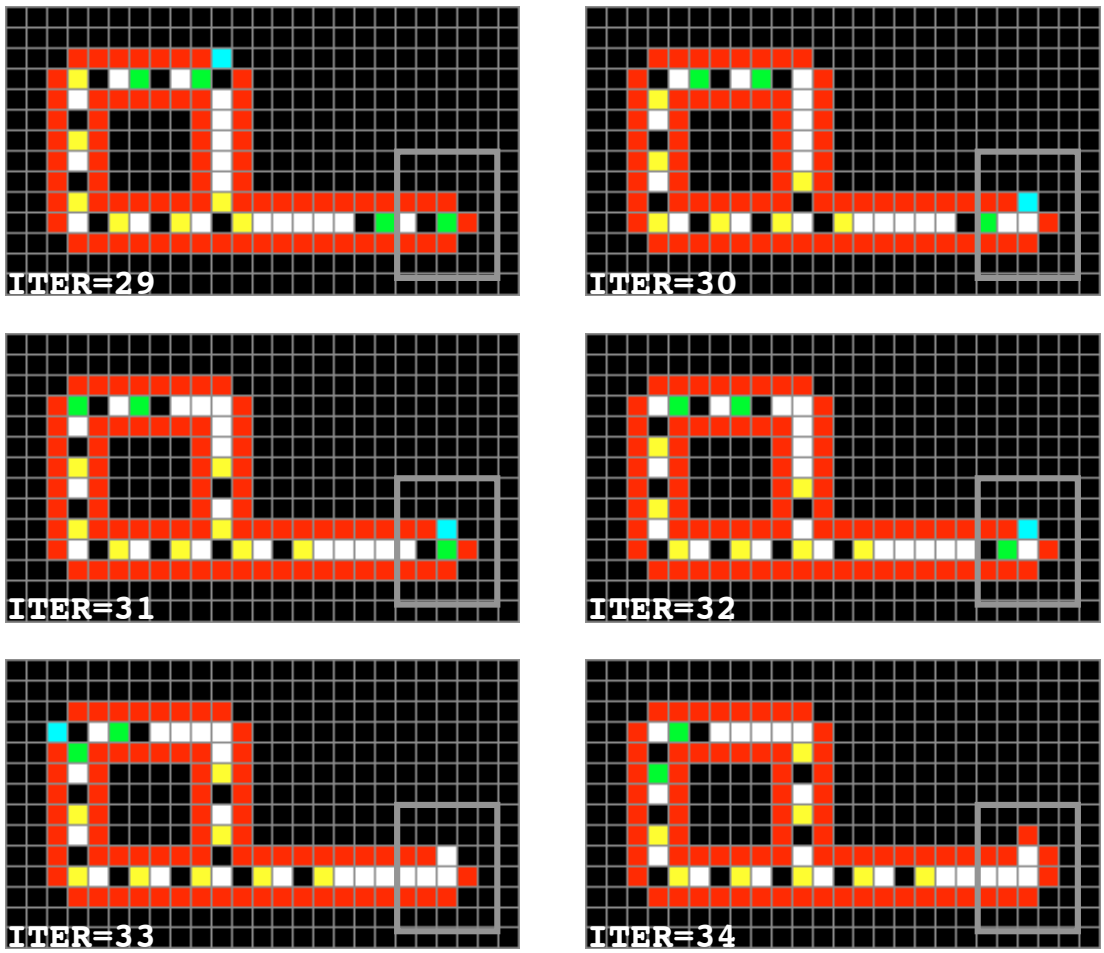

Figure 3-8: The constructing arm turns 90º to the left.

The sequential nature of the self-reproduction process generates a spiraling pattern in the propagation of the loop through space (Fig. 3-10): as each loop tries to reproduce in the four cardinal directions, it finds the place already occupied either by its parent or by the offspring of another loop, in which case it dies (the data within the loop is destroyed).

Langton's loop uses 8 states for each of the 86 non-quiescent cells making up its initial configuration, a 5-cell neighborhood, and a few hundred transition rules (the exact number depends on whether default rules are used and whether symmetric rules are included in the count).

Further simplifications to Langton's automaton were introduced by Byl [18], who eliminated the internal sheath and reduced the number of states per cell, the number of transition rules, and the number of non-quiescent cells in the initial configuration. Reggia et al. [82] managed to remove also the external sheath, thus designing the smallest self-replicating loop known to date. Given their modest complexity, at least relative to von Neumann's automaton, all of the mentioned automata have been thoroughly simulated.

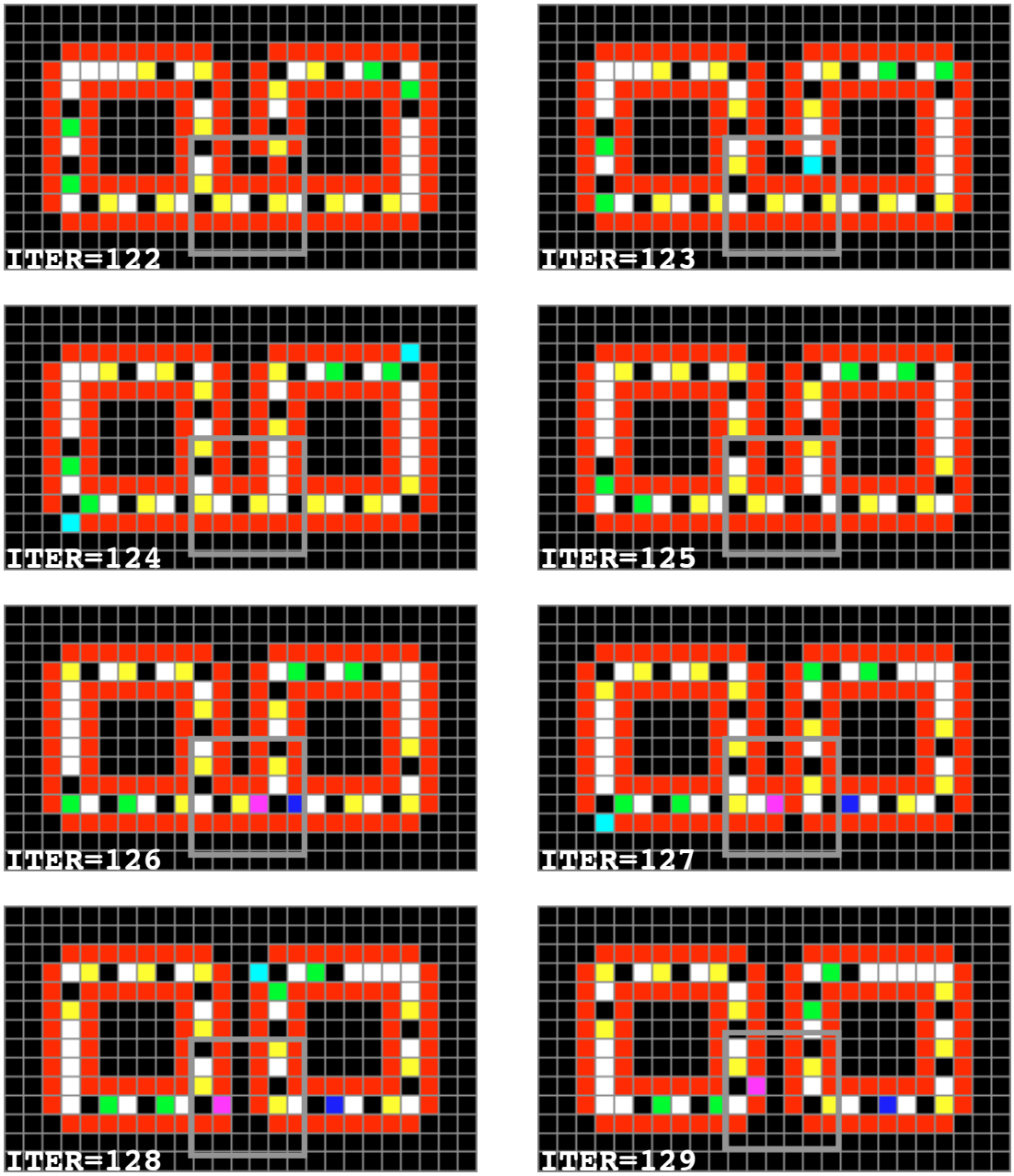

Figure 3-9: The copy is complete and the connection from parent to offspring is severed.

3.4 Self-Replicating Cellular Automata in Embryonics

While the self-replicating automata we introduced in this chapter are indeed interesting examples of self-replicating machines, they do not address some of the requirements of the Embryonics project. In this section, we will attempt to define more precisely these requirements and, after introducing a first attempt to augment the versatility of Langton's loop through the addition of a Turing machine, we will present a new self-replicating automaton we developed in order to offset

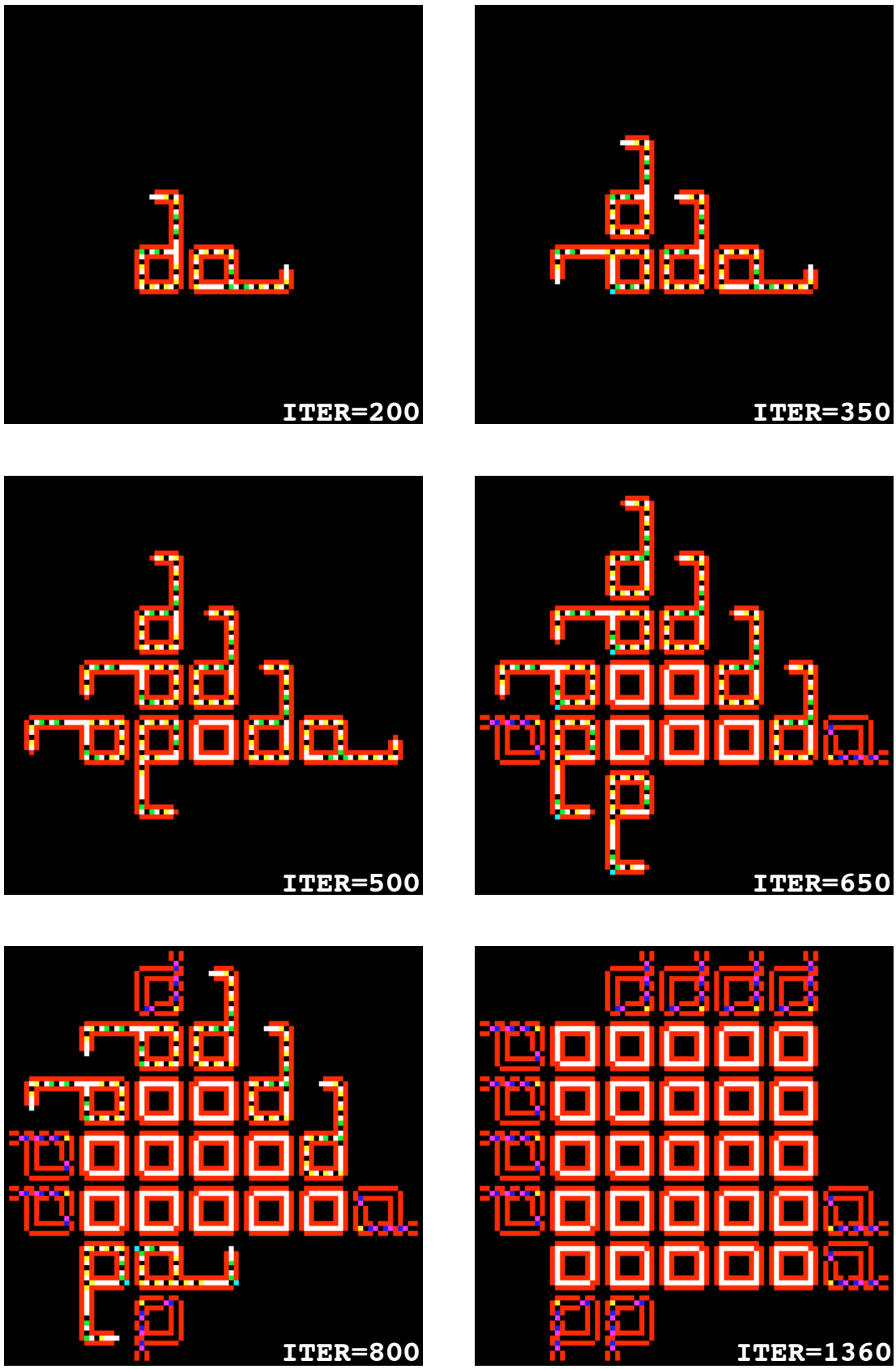

Figure 3-10: Propagation pattern of Langton's loop.

some of the more important deficiencies. We will begin by describing in some detail the operation of the new loop in its most basic configuration, and then illustrate an example of a more elaborate version where a program has been added to the basic automaton to demonstrate its construction capabilities.

3.4.1 The Requirements of the Embryonics Project

Langton's loop represents the best-known example of a simple self-replicating machine, and is therefore of great interest for the Embryonics project. However, it falls short of our requirements in two important aspects:

1. It is designed to operate in an infinite space, whereas the surface of an integrated circuit is necessarily finite. This inconvenience, which might appear relatively minor and easily circumvented, is in fact a major obstacle: the loop's mechanism of self-replication (i.e. the arm) is inherently incapable of handling contact with the array's border.
2. It does not have any functionality beyond self-replication: the loop reproduces and then dies. It is thus more similar to a biological (or even a software) virus than to an actual cell. Once again, the default is structural: because of its origins in the periodic emitter, all the data circulating inside the loop is involved in the generation of the sequence which directs the self-replication process. As we will see, while it is possible to add functionality to Langton's loop, the task is extremely complex and the results not very efficient.

Nevertheless, von Neumann's constructor and Langton's loop are the models which most closely approach our requirements. Therefore, we decided to follow tradition in developing our own approach to self-replication by using cellular automata as an environment to study the issue.

As we already mentioned, there is no formal approach to the development of complex cellular automata. The design of such systems poses therefore considerable problems, since few of the available tools are suited for the task. In particular, no efficient tools were available to help the user in determining the local transition rules necessary to obtain a complex global behavior (cellular automata are mostly used to simulate physical phenomena, where the rules are usually well known). Our first task was therefore to design a software application which would allow us to generate the required rules as easily as possible. The resulting program (described in Appendix A) is, to the best of our knowledge, unique, and proved an invaluable tool in our research.

Equipped with a reasonably powerful tool, we started developing a new automaton capable of self-replication. Considering the complexity of von Neumann's constructor, we decided to draw inspiration from the much simpler Langton's loop, but, as we will see, we had to develop a completely novel mechanism to enable us to circumvent its drawbacks.

3.4.2 A Self-Replicating Turing Machine: Perrier's Loop

As we mentioned, one of the two main problems of Langton's loop is that it is not well adapted to finite CA arrays. Its self-reproduction mechanism assumes that there is enough space for a copy of the loop, and the entire loop becomes non-functional otherwise.

At first glance, this might seem a relatively trivial drawback, which could be overcome by modifying the loop. Such a modification, however, turns out to be very difficult. In fact, to exist in a finite space, and assuming that the automaton has no *a priori* knowledge of the location of the boundaries (a safe assumption, since CA elements have only knowledge of their immediate neighborhood), the loop needs either to verify that enough space is available before it starts the replication process, or else some way to destroy the constructing arm if it detects a boundary during the self-replication process. Either of these mechanisms would require a major structural modification to Langton's loop.

Thus, rather than trying to adapt Langton's automaton to a finite space, we decided to develop an entirely new mechanism, designed specifically to exist in a finite, but arbitrarily large, space, and at the same time capable, unlike Langton's loop, to have a functionality in addition to self-replication.

Adding functionality to Langton's loop, in fact, is not possible without major alterations. As a matter of fact, in the course of our research, we did develop a relatively complex automaton (Fig. 3-11) in which a two-tape Turing machine was appended to Langton's loop [77]. This automaton, developed in our laboratory by

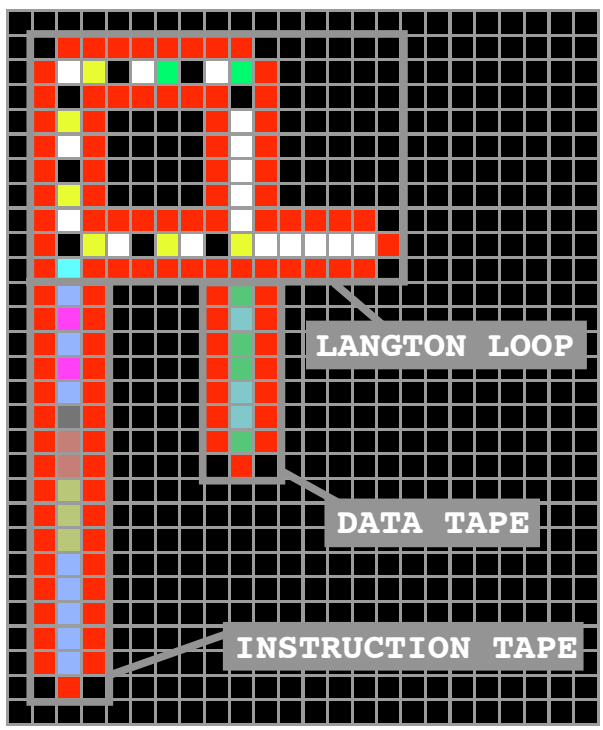

Figure 3-11: A two-tape Turing machine appended to Langton's loop (iteration 0).

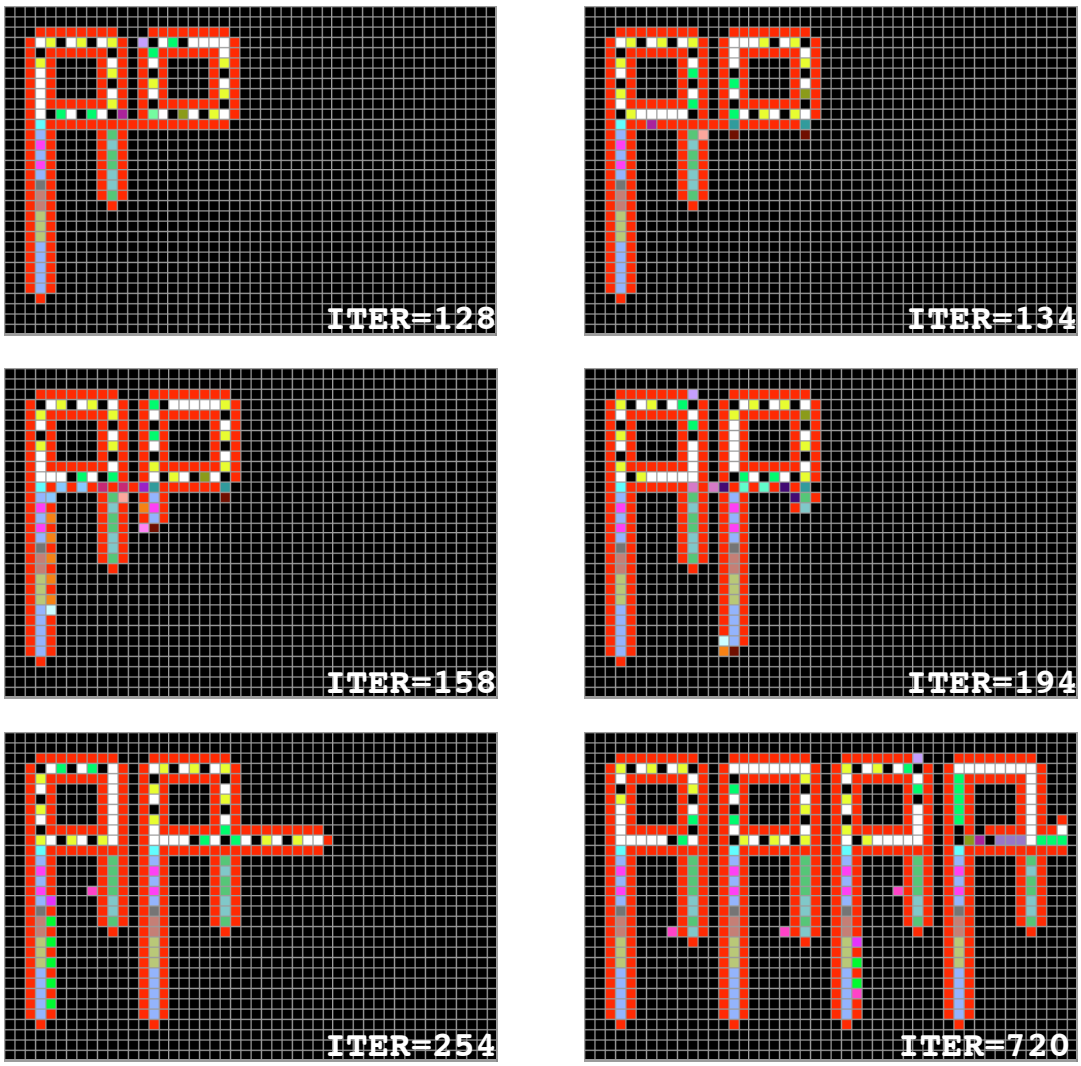

Figure 3-12: Self-replication of the Turing machine.

J.Y. Perrier as a semester-long research project under the supervision of Prof. J. Zahnd, exploits Langton's loop as a sort of "carrier" (Fig. 3-12): the first operation of *Perrier's loop* is to allow Langton's loop to build a copy of itself (iteration 128: note that the copy is limited to one dimension, since the second dimension is taken up by the Turing machine). The main function of the offspring is to determine the location of the copy of the Turing machine (iteration 134). Once the new loop is ready, a "messenger" runs back to the parent loop and starts to duplicate the Turing machine (iterations 158 and 194), a process completely disjoint from the operation of the loop. When the copy is finished (iteration 254), the same messenger activates the Turing machine in the parent loop (the machine had to be inert during the replication process in order to obtain a perfect copy). The process is then repeated in each offspring until the space is filled (iteration 720: as the automaton exploits Langton's loop for replication, meeting the boundary of the array causes the last machine to crash).

The automaton thus becomes a self-replicating Turing machine, a powerful construct which is unfortunately handicapped by its complexity: in order to implement a Turing machine, the automaton requires a very considerable number of additional states (more than 60), as well as an important number of additional transition rules. This kind of complexity, while still relatively minor compared to von Neumann's universal constructor, is nevertheless too important to be really considered for a hardware application. So once again, adapting Langton's loop to fit our requirements proved too complex to be efficient, and we were forced to design a novel automaton to meet our requirements.

3.4.3 A Novel Self-Replicating Loop: Description

In designing our self-replicating automaton [97], we did maintain some of the more interesting features of Langton's loop. In particular, we preserved the structure based on a square loop to dynamically store information. Such storage is convenient in CA because of the locality of the rules. Also, we maintained the concept of constructing arm, in the tradition of von Neumann and his successors, even if we introduced considerable modifications to its structure and operation.

While preserving some of the more interesting features of Langton's loop, we nevertheless introduced some basic structural alterations:

- We use a 9-element neighborhood (the element itself plus its 8 neighbors).
- As in Byl's version of Langton's loop, we use only one sheath, but contrary to Byl, we retain the internal sheath and eliminate the external one. This allows us to let the data in the loop circulate without the need for leading or trailing states (the black and white elements in Langton's loop). In addition to the internal sheath, we have four *gate elements* (in the same state as the sheath) outside the loop at the four corners of the automaton. These elements are initially in the "open" position, and will shift to the "closed" position once the copy is accomplished.
- We extend four constructing arms in the four cardinal directions at the same time, and thus create four copies of the original automaton in the four directions in parallel. When the arm meets an obstacle (either the boundary of the array or an existing copy of the loop), it simply retracts and puts the corresponding gate element in the closed position. This mechanism allows us to overcome the first major drawback of Langton's loop in relation to the Embryonics project (its inability to work properly in a finite space).
- Rather than being directed to advance, our constructing arm advances by default. As a consequence, it is necessary only to direct it to turn at the appropriate moment. This is done by sending periodic "messengers" to the tip of the constructing arm.

- The arm does not immediately construct the entire loop. Rather, it constructs a sheath of the same size as the original. Once the sheath is ready, the data circulating in the loop is duplicated and the copy is sent along the constructing arm to wrap around the new sheath. When the new loop is completed, the constructing arm retracts and closes the gate. As we will see, dividing the self-replication process in two phases is a major asset in the transition to digital hardware.
- As a consequence, we use only four of the circulating elements to generate the messengers. Since the only operation performed on the remaining data elements is duplication, they do not have to be in any particular state. In particular, they can be used as a "program", i.e., a set of states with their own transition rules which will then be applied alongside the self-reproduction to execute some function, allowing us to overcome the second drawback of Langton's loop (its lack of functionality beyond self-reproduction).
- Unlike Langton's loop, our loop does not "die" once duplication is achieved, as the circulating data remains untouched by the self-reproduction process. This feature is a requirement for implementing functions which work after the copy has finished. As a side benefit, it becomes possible to force the loop to try and duplicate again in any of the four directions simply by shifting the corresponding gate back to the open position. This feature could be interesting in view of self-repair: a dead loop can be reconstructed by its neighbors.

As should be obvious, while our loop owes to von Neumann the concept of constructing arm and to Langton (and/or Codd) the basic loop structure, it is in fact a very different automaton, endowed with some of the properties of both.

We have seen that von Neumann's automaton is extremely complex, while Langton's loop is very simple. The complexity of our automaton is more difficult to estimate, as it depends on the data circulating in the loop. The number of non-quiescent elements making up the initial configuration depends directly on the size of the circulating program. The more complex (i.e. the longer) the program, the larger the automaton (it should be noted, however, that the complexity of the self-reproduction process does not depend on the size of the loop). The number of states also depends on the complexity of the program. To the 5 "basic" states used for self-reproduction (see description below) must be added the "data states" (at least one) used in the program, which must be disjoint from the basic states. The number of transition rules is obviously a function of the number of data states: in the basic configuration (i.e., one data state), the automaton needs 692 rules[10] (173 rules rotated in the four directions).

The complexity of the basic configuration is therefore in the same order as that of Langton's and Byl's loops, with the proviso that it is likely to increase drastically if the data in the loop is used to implement a complex function.

10. By default, all elements remain in the same state.

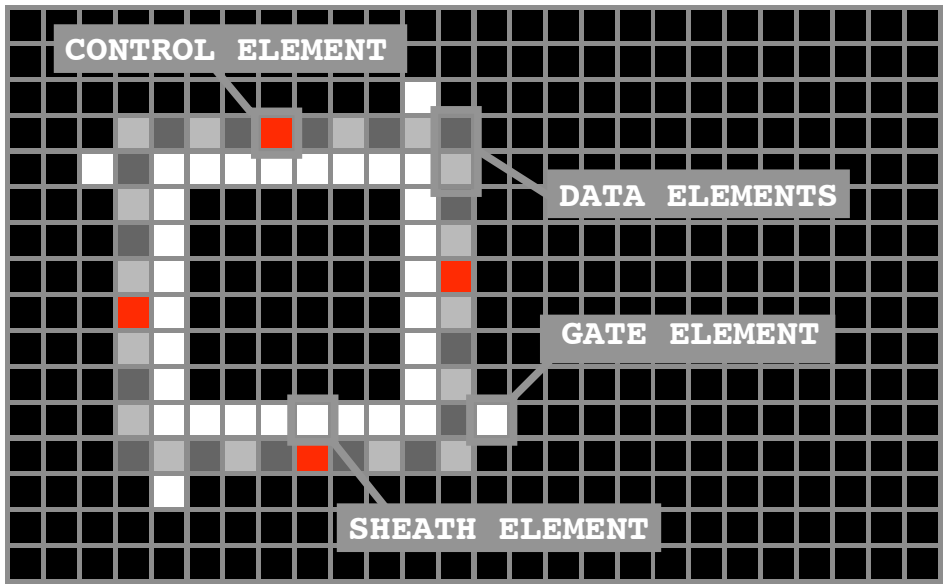

Figure 3-13: The initial configuration of the loop (iteration 0).

3.4.4 A Novel Self-Replicating Loop: Operation

As for von Neumann's and Langton's automata, the ideal space for our automaton is an infinite two-dimensional grid. Since we realize that a practical implementation of such a space might prove difficult, we added some transition rules to handle the collision between the constructing arm and the border of the array. On meeting the border, the arm will retract without attempting to make a copy of the parent loop.

The elements of the array require five basic states and at least one data state (Fig. 3-13). State 0 (black) is the *quiescent state*: it represents the inactive background. State 1 (white) is the *sheath state*, that is the state of the elements making up the sheath and the four gates. State 2 (red) is the *activation state* or *control*

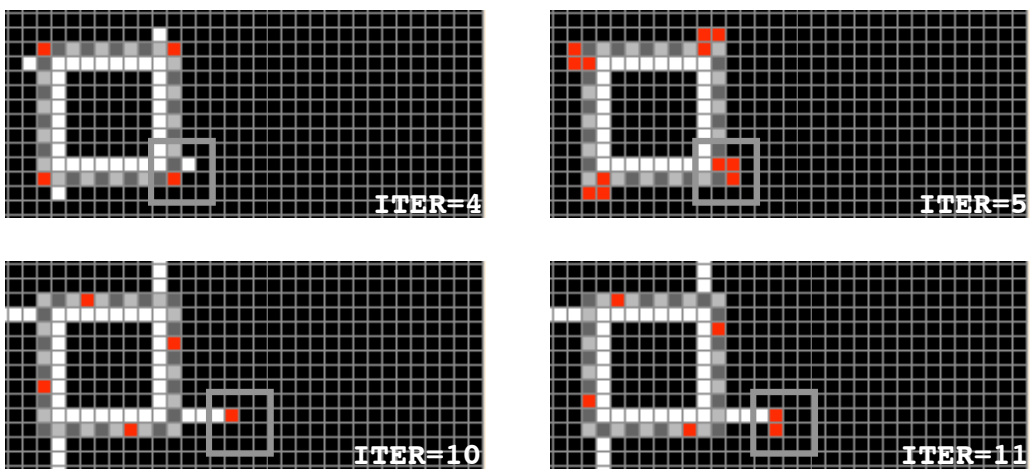

Figure 3-14: The constructing arm begins to extend.

state. The four elements in the loop directing the reproduction are in state 2, as are the messengers which will be used to command the constructing arm and the tip of the constructing arm itself for the first phase of construction, after which the tip of the arm will pass to state 3 (light blue), the *construction state*. State 3 will construct the sheath that will host the offspring, signal the parent loop that the sheath is ready, and lead the duplicated data to the new loop. State 4 (green), the *destruction state*, will destroy the constructing arm once the copy is completed. In addition to these states, two additional *data states* (light and dark grey) represent the information stored in the loop. In this example, they are inactive, while the next section describes a loop where they are used to store an executable program.

The initial configuration is in the form of a square loop wrapped around a sheath. The size of the loop is variable, and for our example is set to 8x8. In the loop is a string of elements of which four are in the activation state (red) and are placed at a distance from each other equal to the side of the loop. Near the four corners of the loop we have placed four elements in the sheath state. These are the gate elements, and the position they occupy at iteration 0 means that the gates are open (that is, that the automaton should attempt to duplicate itself in all four directions).

Once the iterations begin, the data starts turning counterclockwise around the loop. Nothing happens until the first control element reaches a corner of the loop, where it checks the status of the gate. Since the gate is open, the control element splits into two identical elements: the first continues turning around the loop, while the second starts extending the arm (Fig. 3-14). The arm advances by one position every two iterations. Once the arm has started extending, each control element that arrives to a corner will again split and one of the copies will start running along the arm, advancing by one position per iteration (Fig. 3-15). Since the arm is extending at half the speed of the messengers and the messengers are spaced 8 elements apart (the length of one side of the loop), the messengers will reach the tip of the arm at regular intervals corresponding to the length of one side of the loop.

When the first messenger reaches the tip of the arm, the tip, which was until then in state 2, passes to state 3 and continues to advance at the same speed (Fig. 3-16). This transformation tells the arm that it has reached the location of the offspring loop and to start constructing the new sheath.

The next three messengers will force the tip of the arm to turn left (Fig. 3-17), while the fourth will reach the tip as the arm is closing upon itself (Fig. 3-19). It causes the sheath to close and then runs back along the arm to signal to the original loop that the new sheath is ready.

Once the return signal arrives at the corner of the original loop, it waits for the next control element to arrive (Fig. 3-20). When the control element sees the messenger waiting by the gate, once again it splits, one copy staying around the

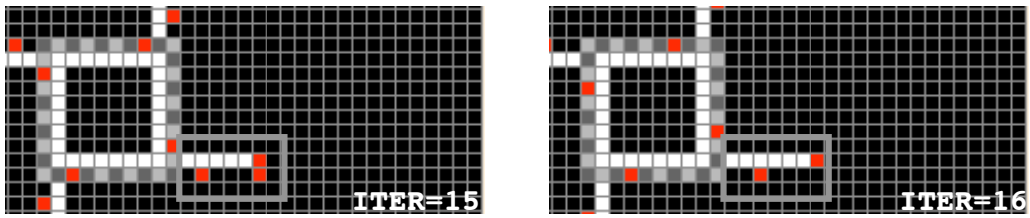

Figure 3-15: The first messenger is running along the arm.

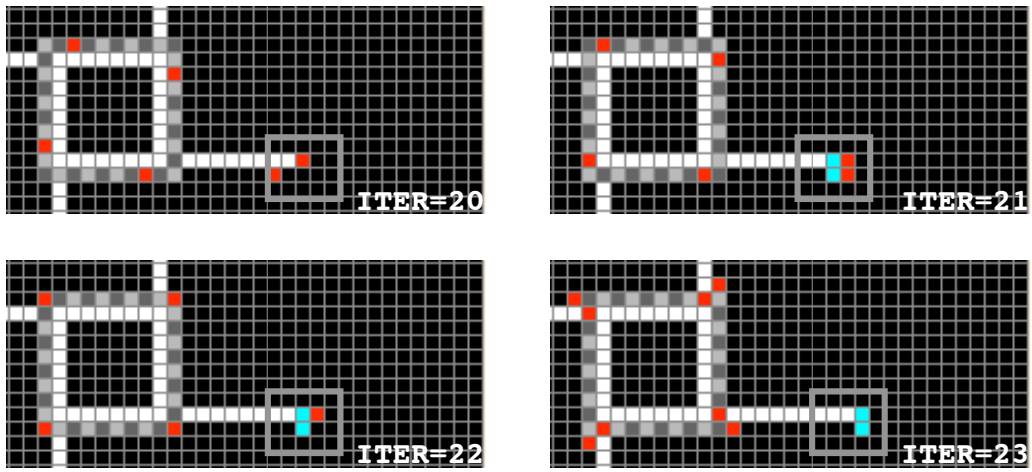

Figure 3-16: The first messenger reaches the tip of the constructing arm.

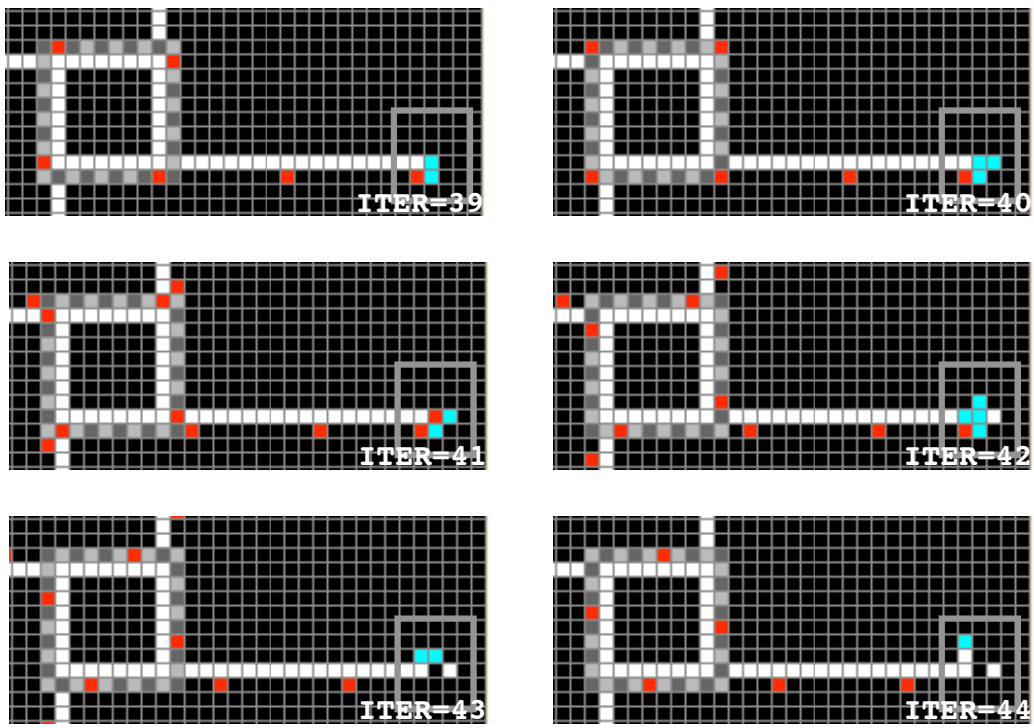

Figure 3-17: The second messenger reaches the tip of the arm, forcing it to turn left.

loop, the other running along the arm. This time, however, rather than running along the arm in isolation as a messenger, it carries behind him a copy of the data in the loop.

Always followed by the data, it runs around the sheath until it has reached the junction where the arm folded upon itself (Fig. 3-18). On reaching that spot, it closes the loop and sends a destruction signal (green) back along the arm. The signal will destroy the arm until it reaches the corner of the original loop, where it closes the gate to avoid further copies.

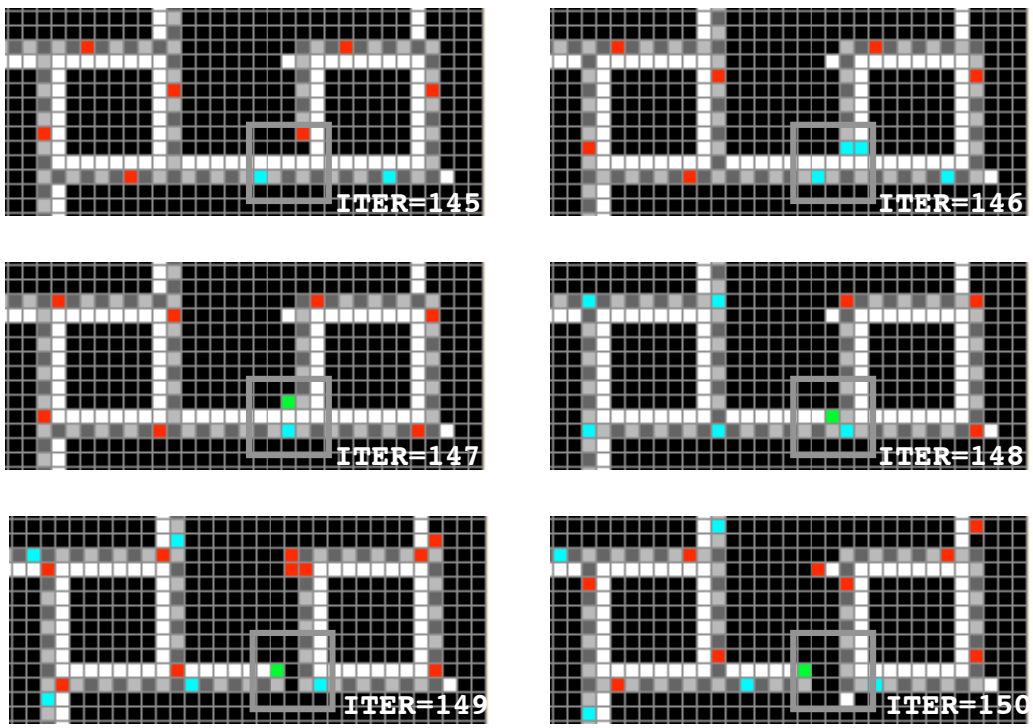

Figure 3-18: The copy is complete and the constructing arm retracts.

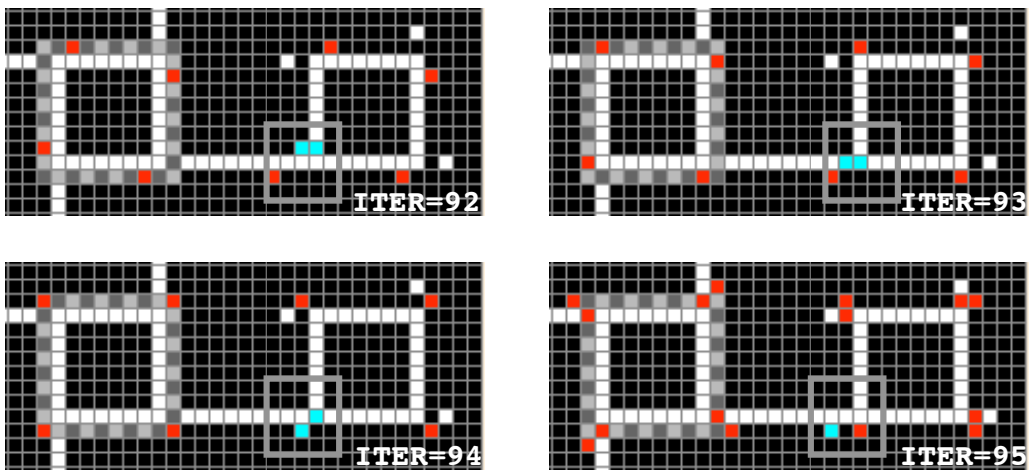

Figure 3-19: The loop is closed, and a new messenger (light blue) is sent back to the parent loop to signal that the offspring is ready to receive the data.

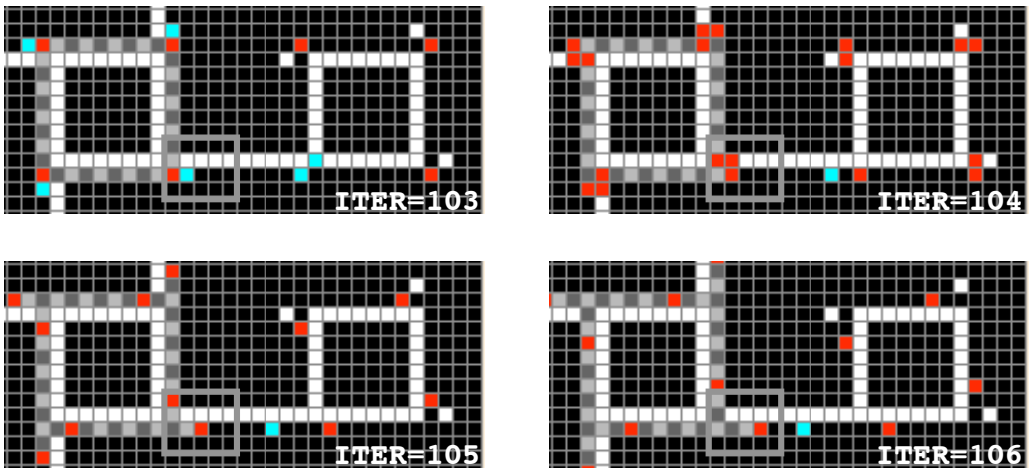

Figure 3-20: A copy of the data is sent from the parent to the offspring.

Meanwhile, the new loop is already staring to reproduce itself in three of the four directions. One direction (down in the figures) is not necessary since another of the new loops will always get there first, and therefore its corresponding gate is automatically set to the closed position. Since the automaton reproduces in all four directions at the same time, its propagation pattern (Fig. 3-21) is somewhat different from that of Langton's loop.

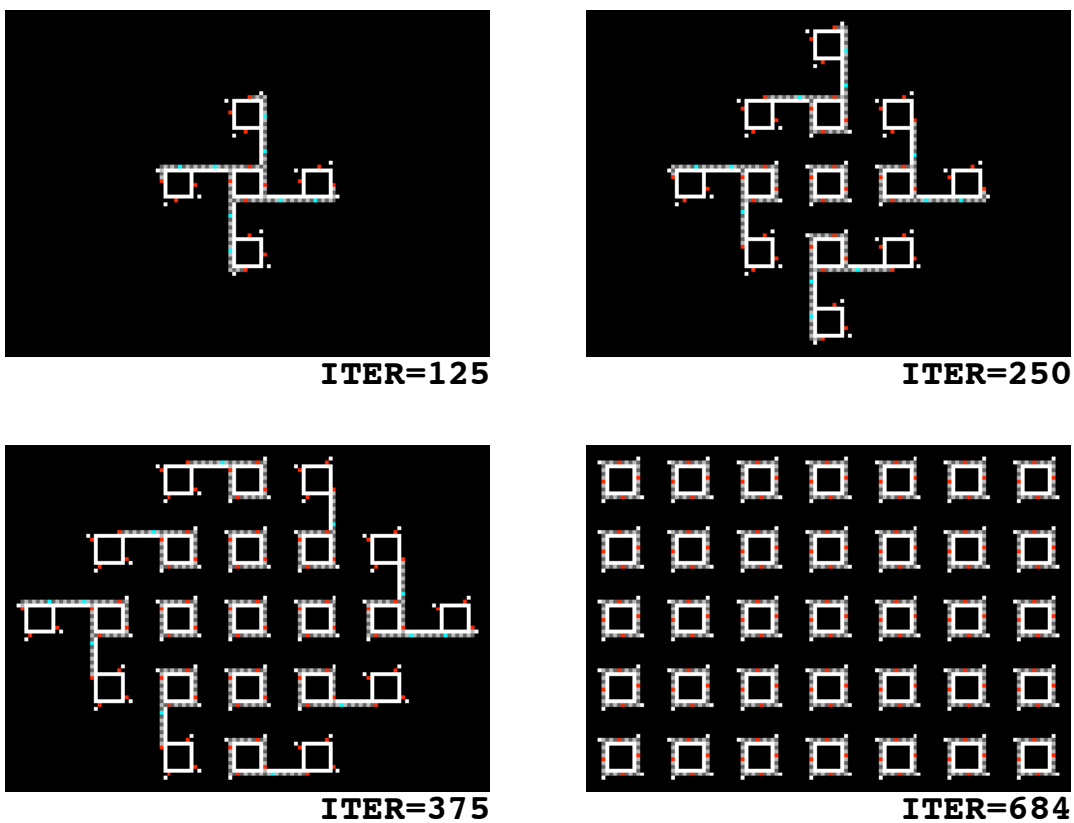

Figure 3-21: The propagation pattern for our loop.

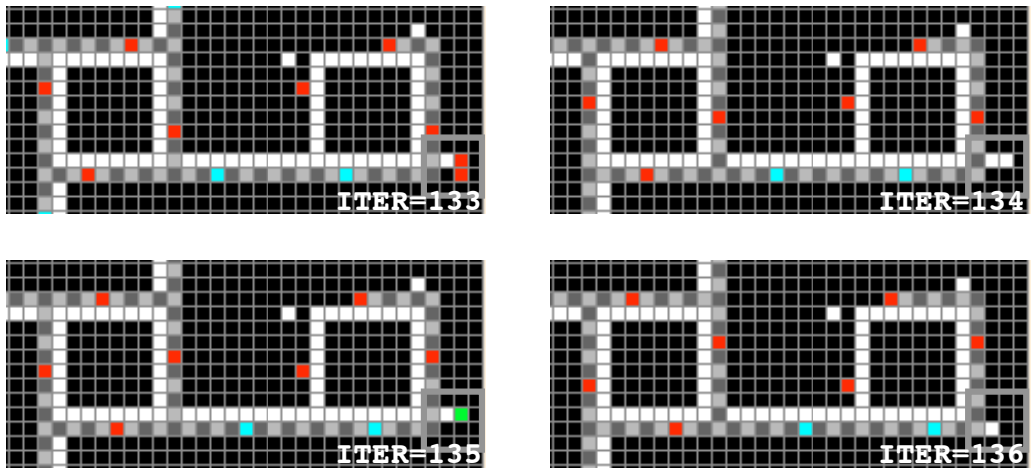

Figure 3-22: The arm, finding the boundary of the array, retracts and closes the gate.

After 121 time periods the gates of the original automaton will be closed and it will enter an inactive state, with the understanding that it will be ready to reproduce itself again should the gates be opened.

The main advantage of the new mechanism is that it becomes relatively simple to retract the arm if an obstacle (either the boundary of the array or another loop) is encountered, and therefore our loop is perfectly capable of operating in a finite space. In the example above, the right border of the figure corresponded to the boundary of the array: when the offspring tried to replicate towards the east, the arm, it found its way blocked, simply retracted and closed its gate (Fig. 3-22).

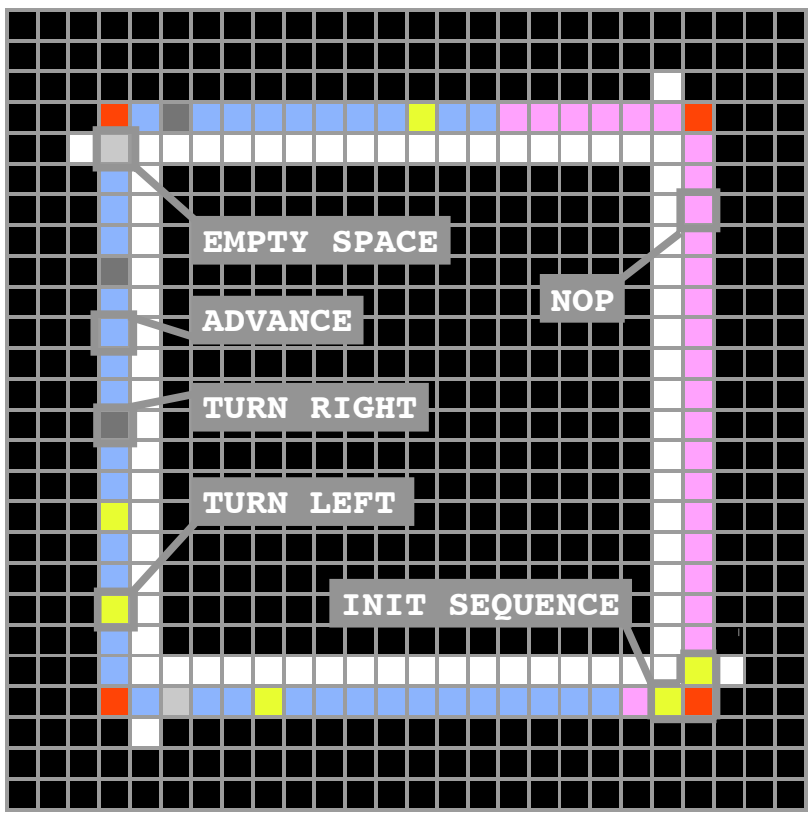

Figure 3-23: Configuration of the LSL automaton at iteration 0.

3.4.5 A Novel Self-Replicating Loop: a Functional Example

In Fig. 3-23, we illustrate an example of how the data states can be used to carry out operations alongside self-reproduction. The operation in question is the construction of three letters, LSL (the acronym of Logic Systems Laboratory), in the empty space inside the loop. Obviously this is not a very useful operation from a computational point of view, but it is a far from trivial construction task which should suffice to demonstrate the capabilities of the automaton.

For this example, we have used 5 data states, each representing an instruction for the construction of the letters: *advance*, *turn left*, *turn right*, *empty space*, and a NOP (no operation) instruction to fill the remaining space in the loop. The construction requires 330 additional rules.

The operation of the program is fairly straightforward. When a certain *initiation sequence* within the loop arrives to the top left corner of the loop, a "door" is opened in the internal sheath (Fig. 3-24). The rest of the program, as it passes by the door in its rotation around the loop, is duplicated and one of the copies is sent to the interior of the loop, where it is treated as a sequence of instructions which direct the construction of the three letters. Once the duplication is complete (i.e., when the first NOP instruction reaches the opening), the door is closed and the sheath reset, except for a flag which indicates that the task has already been completed and prevents the door from being opened again (Fig. 3-25).

The construction mechanism itself is somewhat similar to the method Langton used in his own loop, and is based on a modified constructing arm. The advance instruction causes the arm to advance by one element, the turn left and turn right (Fig. 3-26) instructions cause the arm to change direction, and the "empty space" instruction produce a gap in the arm (to separate the letters).

During the process of reproduction, the program is simply copied (as opposed to interpreted as in the interior of the sheath) and arrives intact in the new loop, where it will execute again exactly as it did in the parent loop (Fig. 3-27).

This is a simple demonstration of one way in which the data in the loop could be used as an executable program. Of course, many other methods can be envisaged, but unfortunately it would be very hard, if not impossible, to obtain computationally interesting self-replicating systems using "pure" cellular automata.

In fact, CA are, by definition, closed systems: all the information must be present in the array at iteration 0 (in our case, all the data for the system must be included in the initial loop). Since useful computation would require that each of the offspring execute a different function (or at the very least, the same function on different data), the requirement that all information be stored in the parent loop is too restrictive for our needs.

At this stage we therefore decided to stop further development of self-replicating machines in the cellular automaton environment, and attempt to transfer the accumulated experience to the design of our FPGA.

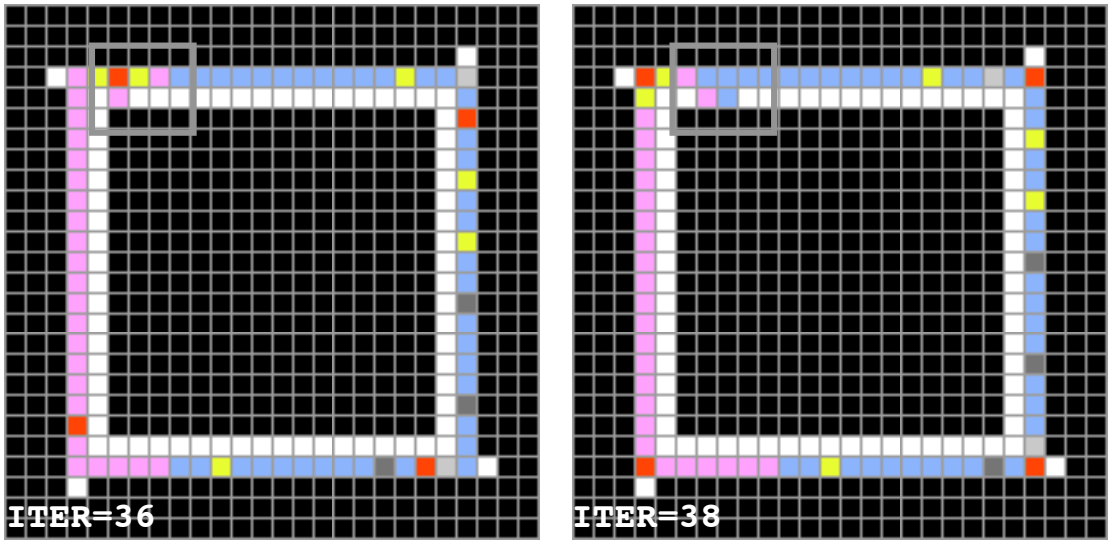

Figure 3-24: The initiation sequence opens a “door” in the sheath.

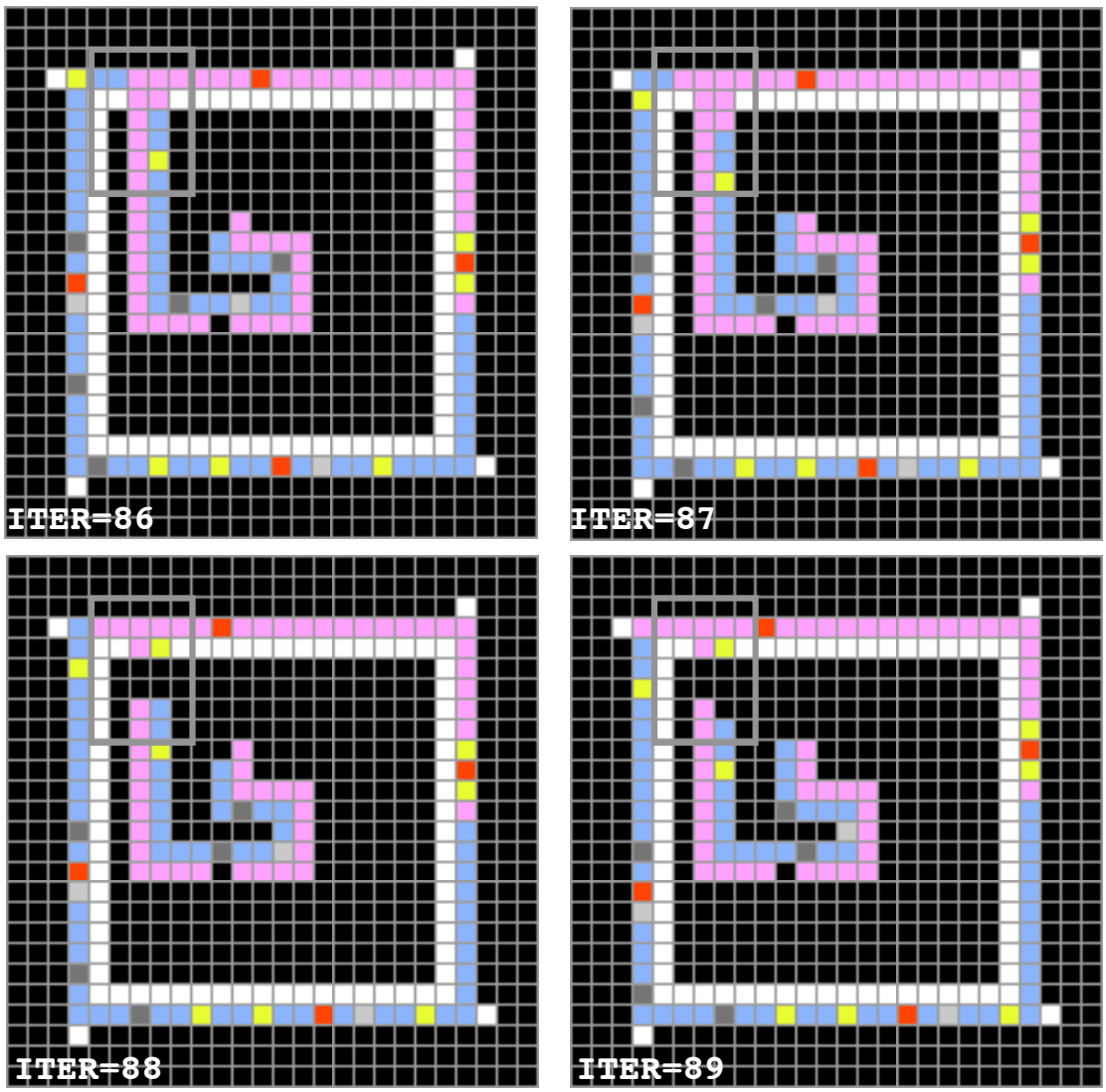

Figure 3-25: The copy of the program is concluded and the “door” is closed.

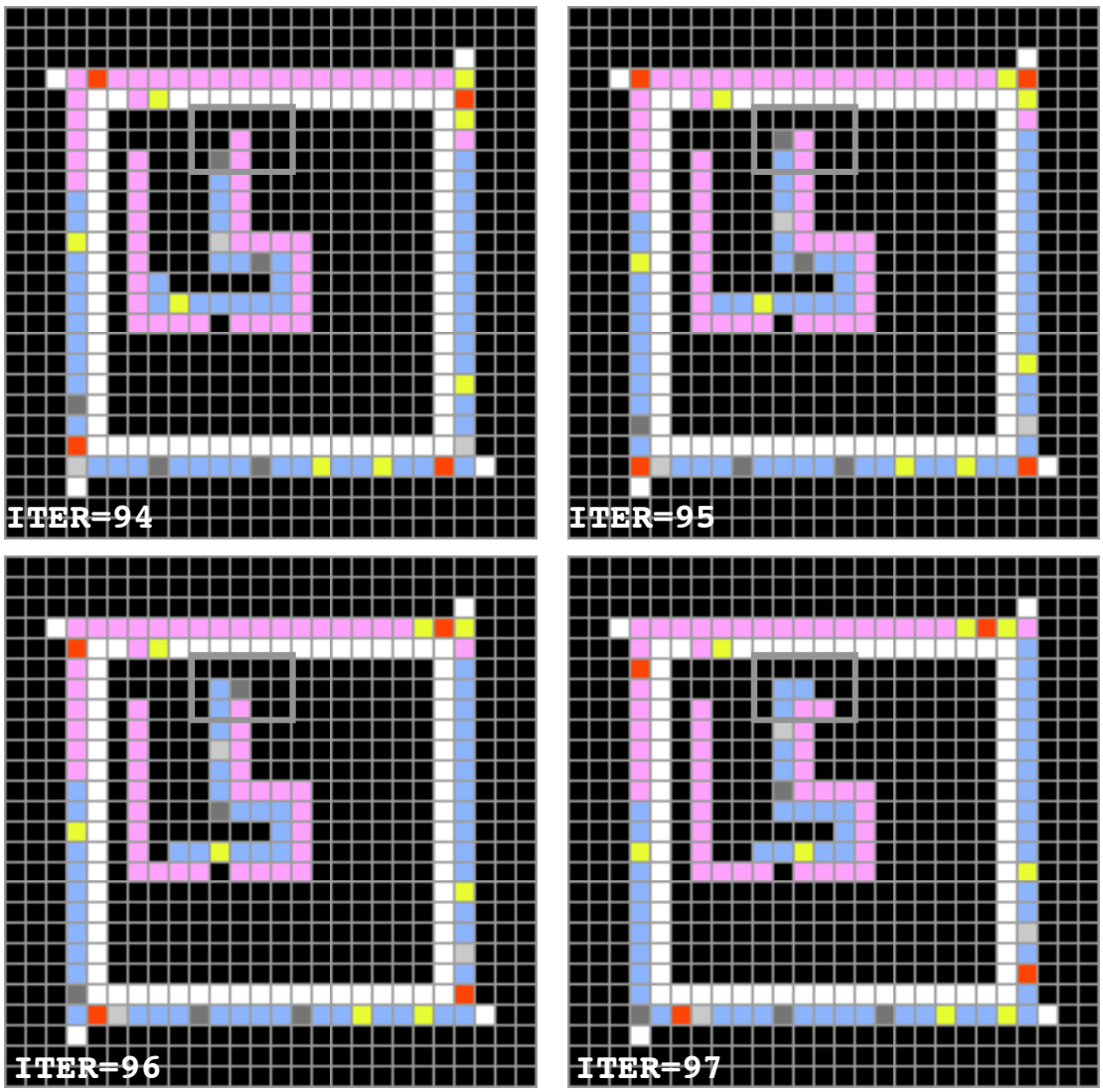

Figure 3-26: The execution of a “turn right” instruction.

3.5 Towards a Digital Hardware Implementation

An FPGA circuit is significantly different from a cellular automaton, a certain superficial resemblance notwithstanding. To develop a self-replication mechanism which can be efficiently adapted to digital electronic circuits in general and to FPGAs in particular, we therefore had to analyze the operation of our loop and attempt to extract not so much the precise mechanism used to achieve self-replication, but rather the general approach to the problem. In this section, we will present the results of this process, which led to the development of the *membrane builder*, a very simple cellular automaton which we then implemented in hardware and integrated into our FPGA.

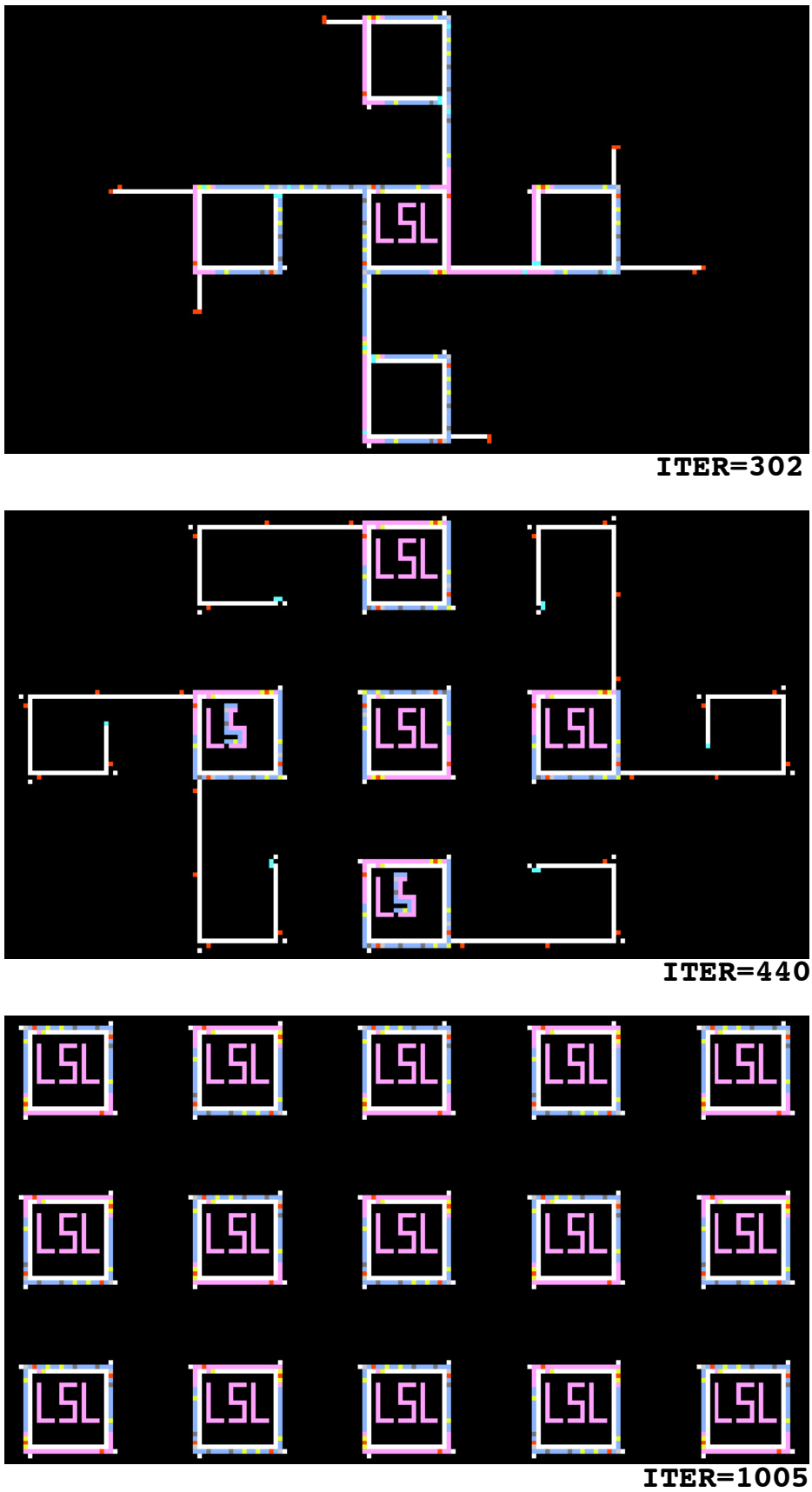

Figure 3-27: The program is copied and executed into each of the offspring.

3.5.1 The Membrane Builder

Our automaton fits most of our requirements: it is a computing machine (it would be possible, if difficult, to add a universal Turing machine to the loop) capable of self-replication. The transition from cellular automata to hardware, however, requires a process of synthesis: CAs, as we have seen, are very inefficient from the point of view of an hardware realization (every element needs to access a substantial lookup table), hard to design (even with our dedicated tool, the design of a complex automaton remains a daunting endeavor), and fragile (a single faulty element almost invariably destroys the entire system). To find a viable approach to the realization of self-replicating hardware, we thus had to extract from our automaton the essential features of the self-replicating mechanism, and use them as a basis for the design of an electronic circuit.

One of the main differences between our loop and Langton's lies in the two-phase mechanism of self-replication: while in Langton's loop the process is indivisible, we can identify two distinct phases in the self-replication of our loop: first the arm "reserves" the space required for the offspring, and only then the data are copied. In other words, the process can be divided into a *structural phase*, where the structure of the loop is set into the empty space, and a *configuration phase*, where the functionality of the parent is sent into the offspring.

In defining an FPGA architecture to implement the process of self-replication, the requirements of cellular automata conflict with an efficient use of hardware in a number of respects, notably where the configuration phase is concerned: the loop structure, so well suited to cellular automata, represents an unacceptable waste of material in an electronic circuit (using the loop's perimeter implies that the interior is wasted). As we will see in the next chapter, to efficiently exploit the surface of programmable logic we adopted a more conventional approach to the configuration of the array. However, the separation between the two phases of our self-replication strategy means that the drawbacks of the configuration phase need not necessarily apply to the structural phase.

The greatest obstacle in implementing self-replication in our FPGA lies in the fact that we cannot *a priori* define the size of our cells. Thus, we require a system capable of configuring a finite two-dimensional array of elements (the FPGA) as a two-dimensional array of identical configuration *blocks*, each containing a single cell. Obviously, this problem bears a considerable resemblance to self-replication in cellular automata: similarities exist between the CA elements and the FPGA elements (molecules) and between a loop and a processor (cells). By exploiting this resemblance, we can observe that the structural phase of our self-replication mechanism can indeed be a solution to the problem of configuring our FPGA.

If we consider the FPGA before configuration as an array of CA elements in the quiescent state, we can design a very simple automaton [60, 92] (Fig. 3-28) capable of subdividing the surface of programmable logic into square blocks of variable size (we thank André Stauffer for the realization of this automaton). The initial configuration (iteration 0) of this automaton is somewhat non-conven-

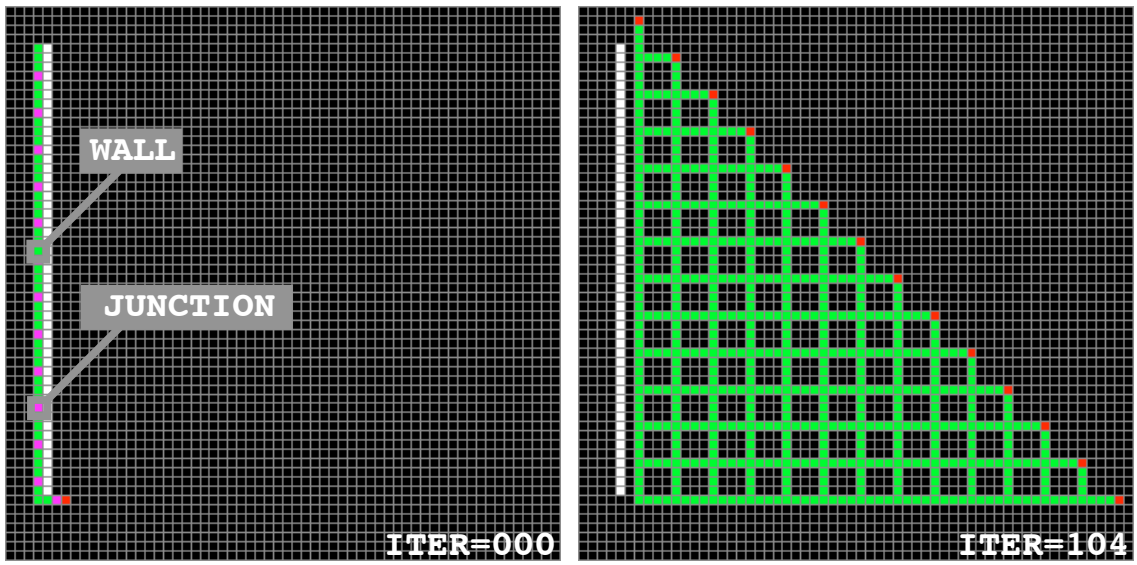

Figure 3-28: An extremely simple cellular automaton capable of subdividing a two-dimensional array from a one-dimensional description.

tional: the configuration of an FPGA is effected by a one-dimensional sequence of bits (the configuration bitstream), while conventional cellular automata operate from an initial two-dimensional structure already in place at iteration 0. To overcome this difference, we designed an initial configuration which *simulates* a configuration stream: the sequence of states stored alongside the (inert) white band represents the data stored in a memory, and as time progresses it slides along the band to enter the quiescent area (the empty FPGA) from its lower left corner.

The mechanism used in this automaton is very similar to the one used to direct the constructing arm in our loop, but is much simpler. It uses only two active states[11]: a *wall state* (green), which defines the borders of the blocks, and a *junction state* (purple), which defines the intersections between walls. Operation starts when the first data element (always in the junction state) leaves the inert band. From this first element, two branches start to propagate: one to the north and one to the east. The branches will advance at half speed, leaving walls on their path. Meanwhile, data is leaving the band at full speed. Whenever a junction state reaches the tip of the branches, it causes a further split: again, one branch to the north and one to the west. It should then be obvious that in the end, the quiescent area will have been divided into square blocks of a size corresponding to the distance between two junction states in the data tape.

This very simple cellular automaton could thus allow us to achieve self-replication in an FPGA by transforming a one-dimensional string of information to a two-dimensional array of square blocks of programmable size: the elements of the cellular automaton form the perimeter of the block, and thus perform the same function as the *membrane* which surrounds a biological cell. Moreover, the automaton is simple enough that a hardware implementation becomes straightforward.

11. Some additional states are indeed required for the operation of the automaton, but they are either inert (e.g., the black quiescent state and the white band) or transient.

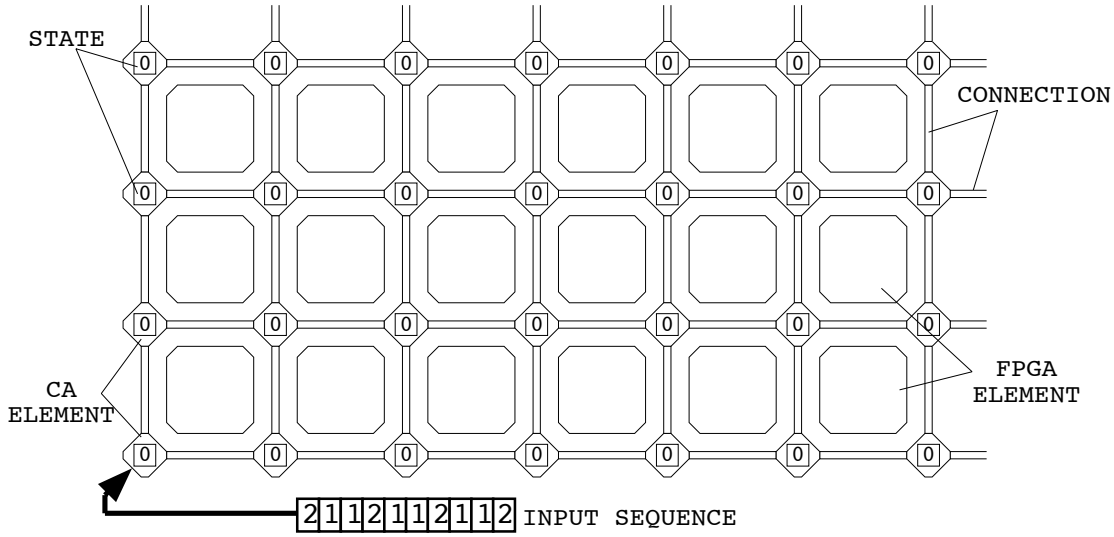

Figure 3-29: The CA, set into the FPGA, can be programmed with a sequence of states.

3.5.2 A Self-Replicating FPGA

The next step in implementing a self-replicating FPGA is to integrate the automaton into the two-dimensional array of programmable elements. After experimenting with different alternatives, we decided to insert the CA elements in the space between the FPGA elements (Fig. 3-29). The configuration stream enters the array in the southwest corner, and propagates through the automaton: this operation is completely independent of the structure of the programmable elements, and can thus be applied to any FPGA architecture.

Fig. 3-30 shows the effect of entering the simple state sequence shown in Fig. 3-29 into the automaton. As for the automaton described above, only two active states are used: the wall state, labeled `1`, and the junction state, labeled `2`. The operation of this automaton is identical to that of the one shown in the previous subsection, with a single exception: the corners of the blocks remain in a distinctive junction state after the end of the propagation. This small difference, which in fact allows us to reduce the complexity of the hardware, is a typical example of the difficulties of designing a cellular automaton for hardware implementation: an increase in the complexity of the automaton can, as often as not, result in a decrease in the complexity of the hardware required to implement it.

In the end, we were able to design an extremely simple mechanism which allows us to partition the FPGA into a set of square blocks of programmable size (shown in Fig. 3-31 with a more intuitive symbolic representation of the CA states), the indispensable preamble to self-replication. Once the membrane is in place, we can, as we will see in the next chapter, use it to direct the configuration of the FPGA: we can exploit the information contained in the cellular automaton to automatically replicate the configuration of a single block into all of the blocks in parallel. We can, in short, achieve the self-replication of electronic circuits[12].

12. The process also resembles the biological process of *cloning*. It is self-replication in the sense that it is handled by the hardware itself, and it is cloning in the sense that all data come from an external source.

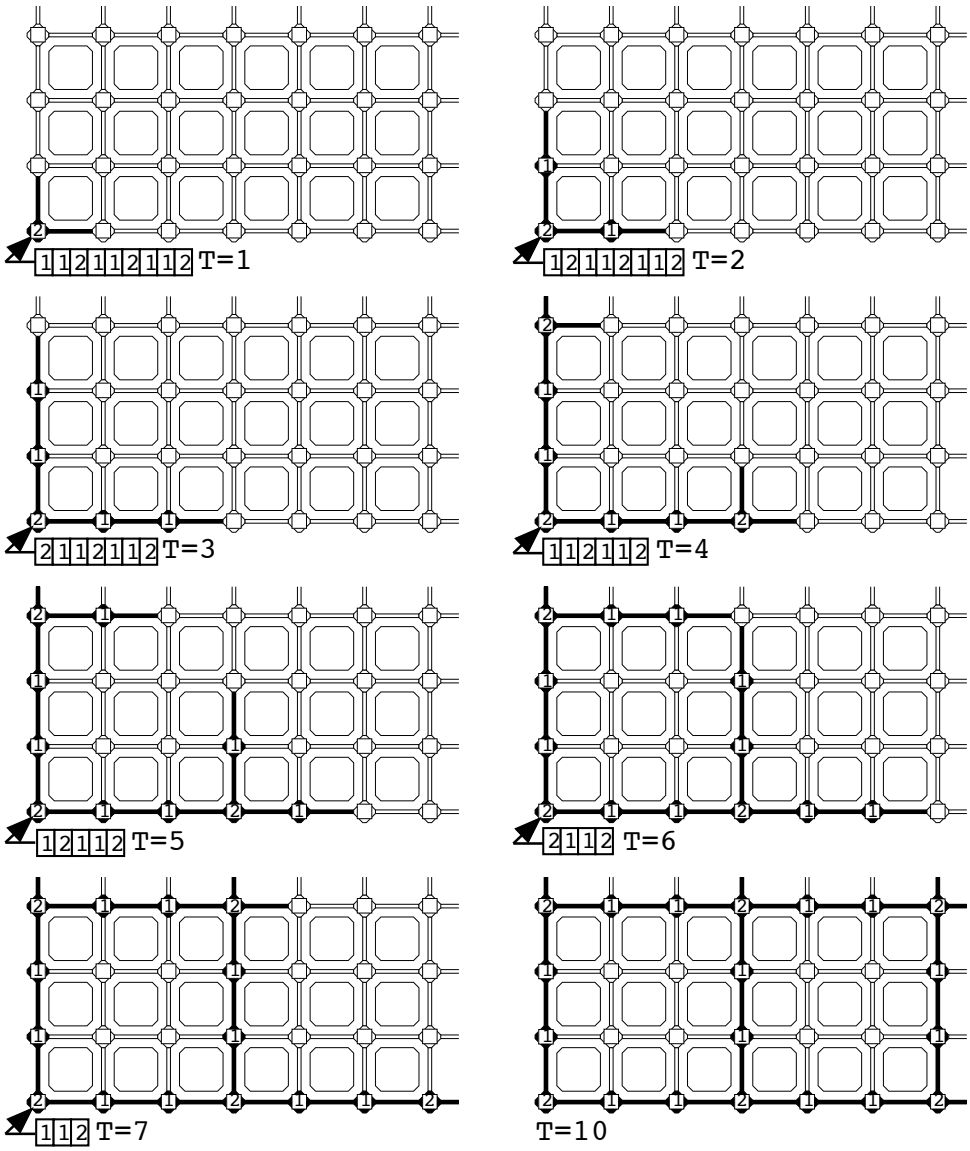

Figure 3-30: The cellular automaton partitions the array in the desired blocks.

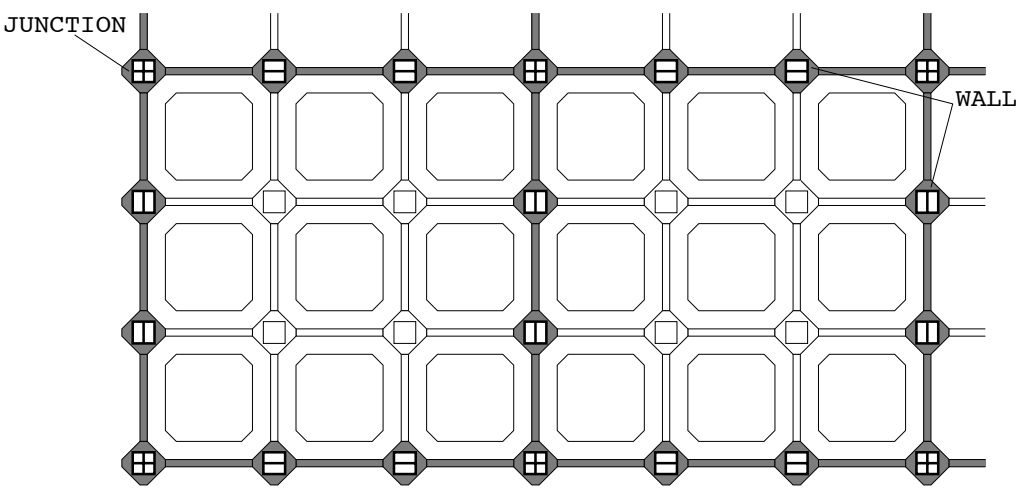

Figure 3-31: The FPGA after partitioning, ready for self-replication.

CHAPTER 4

SELF-REPAIR

Having defined a first, acceptable solution to the problem of implementing self-replication in our FPGA, we turned to the second bio-inspired feature required by our system: *self-repair* (in other words, healing).

This chapter will describe our approach to the design of a self-repairing FPGA. Section 4.1 will provide a detailed description of MuxTree, the multiplexer-based FPGA we developed to implement the molecular level of our ontogenetic system. Section 4.2 will discuss the problem of *self-test*, an essential preliminary to self-repair. Section 4.3 we will proceed to describe the reconfiguration mechanism we adopted to implement self-repair, and introduce a modified self-replication mechanism, containing a set of novel features to provide support for self-repair. In the last section (4.4) we will provide a simple (but complete) example of the behavior of our FPGA in the presence of faults.

4.1 A New Multiplexer-Based FPGA: MuxTree

Like all FPGAs, MuxTree [24, 60, 61, 98] is a two-dimensional array of elements (the molecules of our three-level system) which, in MuxTree's case, are particularly small. Each element, in fact, is capable of implementing a universal function of a single variable and of storing a single bit of information. As we will see, the small size can be both an advantage and a disadvantage, depending on the intended application. In the next subsections, we will describe in detail the basic element (Fig. 4-1), which will be divided into three parts: the programmable function (`FU`), the programmable connections (`SB`), and the configuration register (`CREG`), and then discuss the positive and negative aspects of such an architecture.

4.1.1 The Programmable Function

The complexity, or *grain*, of an FPGA element can vary considerably from one architecture to the next. The only actual requirement is that it must be possible to implement any given function using one or more elements. In addition, it is customary, if not strictly required, to include some form of memory in an element so as to be able to easily implement sequential systems.

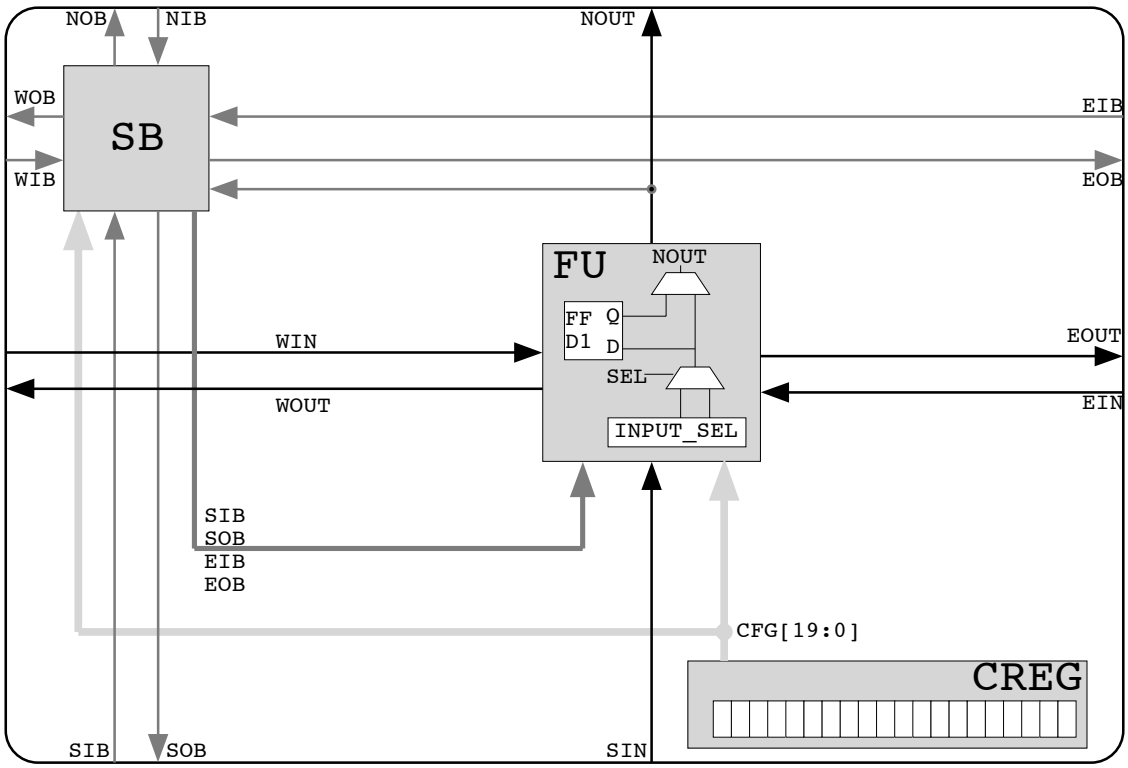

Figure 4-1: Overall structure of a MuxTree element.

MuxTree is no exception to this rule, but is unusual in that it is remarkably fine-grained: the programmable function is realized using a single two-input multiplexer (Fig. 4-2). The multiplexer being a universal gate (i.e., it is possible to realize any function given a sufficient number of multiplexers), the first requirement for an FPGA element is respected. Each element is also capable of storing a single bit of information in a D-type flip-flop, fulfilling the second requirement.

The programmable function is therefore realized by the single multiplexer `M0`. The two one-bit-wide inputs are programmable, and 6 bits of the element's configuration (`LS[2:0]` for the left input and `RS[2:0]` for the right input) are used to select two of eight possible choices:

- the constant logic value `0`;
- the constant logic value `1`;
- the output of the element immediately to the south (`SIN`);
- the output of the element to the southeast (`EIN`);
- the output of the element to the southwest (`WIN`);
- the output of the flip-flop `F`;
- the long-distance connection `SIB` (see the next section);
- the long-distance connection `SOB` (see the next section).

The source of the multiplexer's control variable is also programmable: a single bit of the configuration (`M`) selects whether the value of `EIB` or that of `EOB` (both long-distance connections, as explained below) will control the multiplexer.

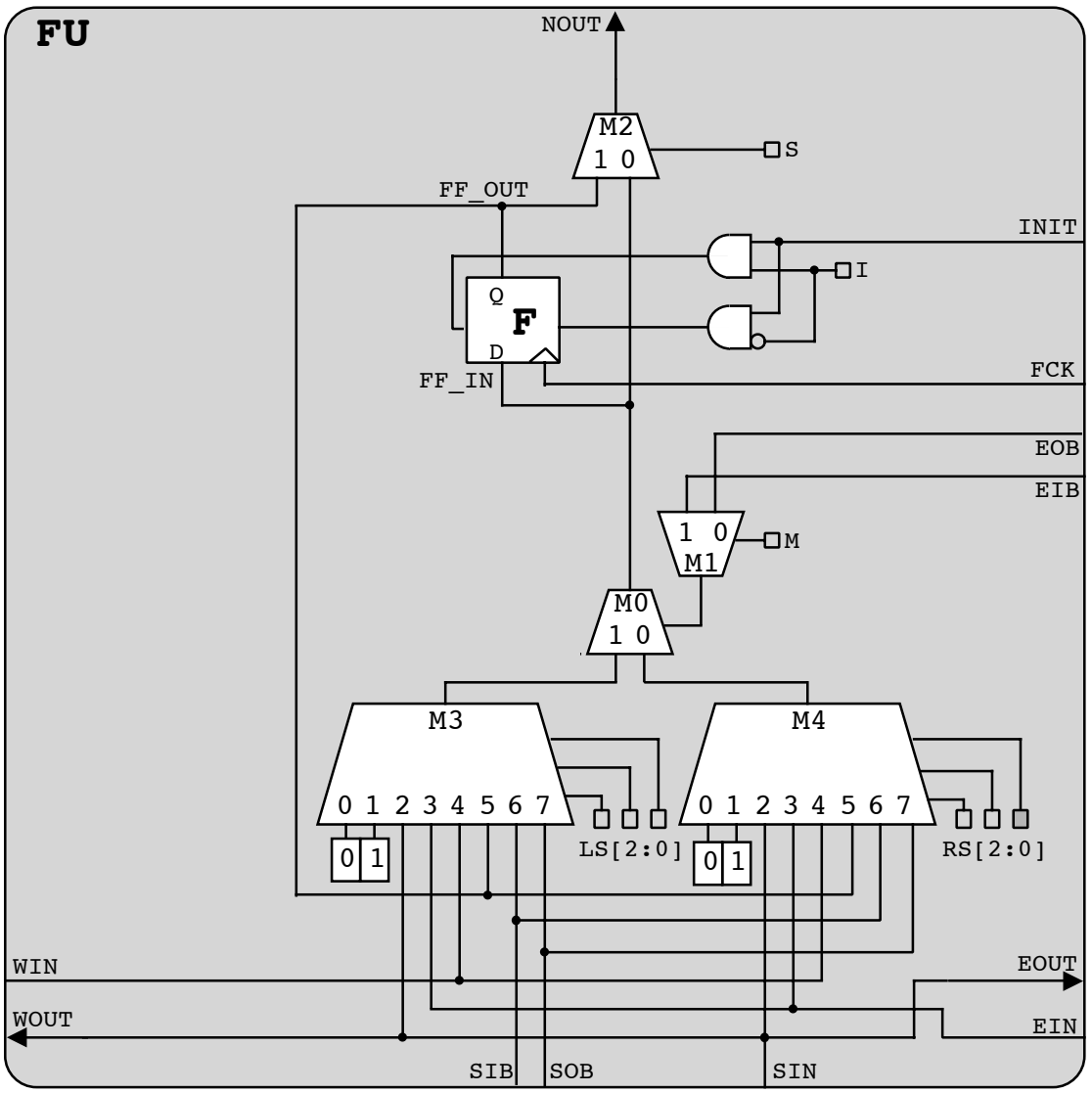

Figure 4-2: The programmable functional unit FU of a MuxTree element.

The output NOUT of the element can have two sources, depending on the value of configuration bit S. If the element is purely combinational, S=0 and NOUT is the output of the multiplexer. If, on the other hand, the element is supposed to have a sequential behavior, S=1 and NOUT propagates the output of the flip-flop F.

The D-type flip-flop F is therefore used to implement sequential behavior. Its purpose is to store the output of the multiplexer at the rising edge of a functional clock FCK, whose period depends on the application[1]. The configuration bit I allows the user to define a default value for F, which will be restored by the initialization signal INIT.

1. As in any electronic circuit, the maximum frequency in an FPGA depends on the longest combinational path in the array. Since the connections of an FPGA are programmable, the longest path changes with each configuration.

4.1.2 The Programmable Connections

There are two separate sets of connections in a MuxTree element: a fixed short-distance network (in black in Fig. 4-1) for communication between immediate neighbors, and a programmable long-distance network (in dark gray) to allow for data to be exchanged between more distant elements.

The first (short-distance) network consists of the following connections:

- an input line from the neighbor to the south (`SIN`);
- an output line to the neighbor to the north (`NOUT`);
- an input line from the neighbor to the west (`WIN`), which in practice carries the output of the southwest neighbor;
- an output line to the west neighbor (`WOUT`), which in practice propagates the output of the neighbor to the south;
- an input line from the east neighbor (`EIN`), which in practice carries the output of the southeast neighbor;
- an output line to the east neighbor (`EOUT`), which in practice propagates the output of the neighbor to the south.

Obviously, this short distance network, being fixed (i.e., not altered by the configuration of the element) imposes a certain pattern of communication: each element can access the output of its neighbors to the south, southeast, and southwest, and propagate its own output to its neighbors to the north, northeast, and northwest. This pattern, which might at first sight appear somewhat peculiar, has in fact a definite purpose, as we will explain below in subsection 4.1.4.

Of course, a fixed pattern to an FPGA's connections is a very restrictive limitation. In order to provide a way for communication to occur outside the fixed network, we introduced a second set of connections. This new network (dark gray in Fig. 4-1) is entirely separate from the first, and allows the output of an element to propagate beyond its immediate neighborhood.

The long-distance network provides one input and one output line in each of the four cardinal directions (in Fig. 4-1, `SIB`, `NIB`, `EIB`, and `WIB` are the four input lines, while `SOB`, `NOB`, `EOB`, and `WOB` are the corresponding output lines). The values propagated through these lines are selected in the switch block `SB` (Fig. 4-3), which is controlled by the element's configuration.

The switch block itself consists simply of four multiplexers: for each output, two bits of the configuration determine which of four possible inputs will be selected. Each output line (for example, `EOB`) can thus be connected to an input line coming from the *other* three cardinal directions (`WIB`, `SIB`, or `NOB`), or else to the output `NOUT` of the element. Connections between *any* two elements in the array can thus be realized by routing signals through the switch blocks of the intervening elements, allowing the existence of long-distance connections while preserving the homogeneity of the array (a difficult proposition when more "conventional" hierarchical bus structures are used).

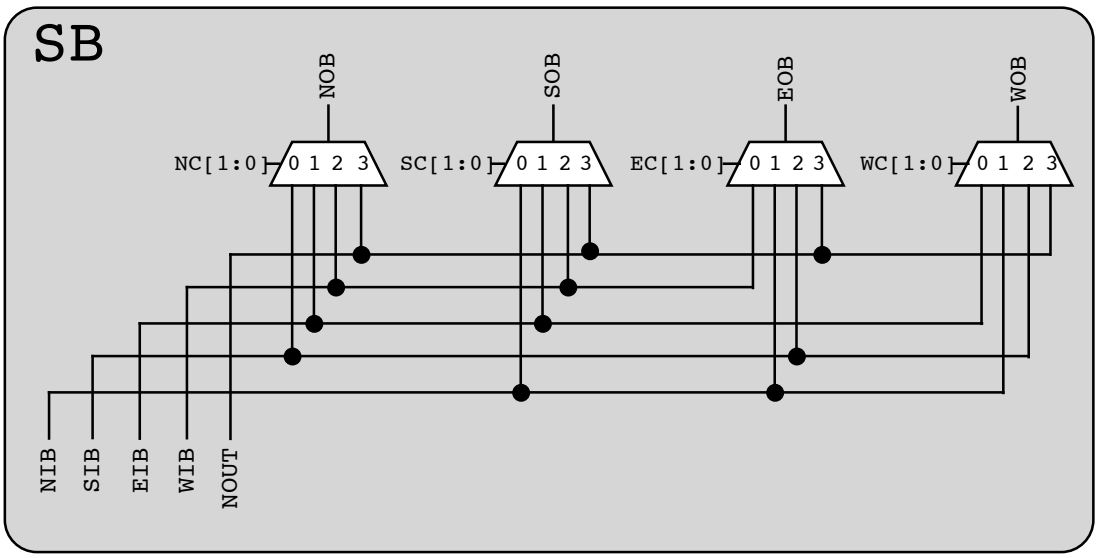

Figure 4-3: The switch block SB which controls the long-distance connections.

The purpose of the long-distance network is to propagate the output NOUT to destinations which are inaccessible through the short distance network. There, this value can be used as either an input to the multiplexer (though lines SIB and SOB) or as the multiplexer's control variable (though lines EIB and EOB).

4.1.3 The Configuration Register

The element's function and connections are determined by a 17-bit configuration, stored in the 20-bit shift register CREG (Fig. 4-4) and structured as follows:

- Bit 00 (the *head* H of the register) stores M, the control variable for multiplexer M1 (Fig. 4-2);
- Bit 01 stores S, the control variable for multiplexer M2 (Fig. 4-2);
- Bit 02 stores I, the default value for flip-flop F (Fig. 4-2)
- Bit 03 is unused in the current implementation;
- Bits 04 to 11 store the control variables for the four multiplexers in the switch block SB (Fig. 4-3);
- Bits 12 to 14 select the left input of multiplexer M0;
- Bit 15 is unused in the current implementation;
- Bits 16 to 18 select the right input of multiplexer M0;
- Bit 19 (the *tail* T) is unused in the current implementation.

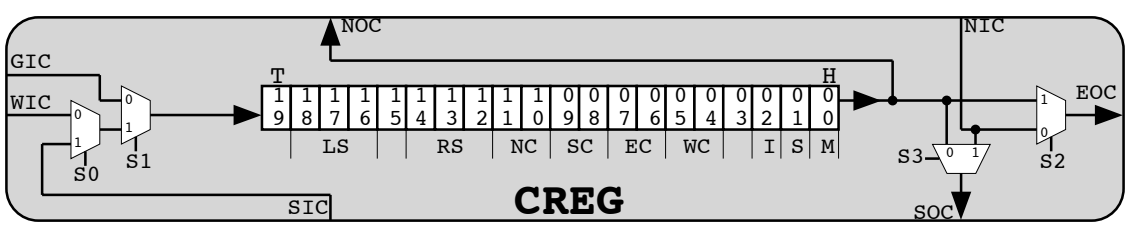

Figure 4-4: The 20-bit register containing the element's configuration.

The configuration of the circuit (i.e., the sum of the configurations of all the elements) is propagated within each of the blocks defined by the self-replication mechanism: all the registers are *chained* together to form one long shift register following a path determined by the block's membrane (Fig. 4-5). The circuit's configuration can thus be seen as a serial bitstream which follows a predetermined path to fill all the elements within a block.

Of course, the definition of such a path requires that a set of programmable connections be provided in order to guide the bitstream to the correct registers. A very simple mechanism, based on a set of four 2-input multiplexers (Fig. 4-4), is sufficient to define a propagation path for the configuration independently of the size of the blocks. The four multiplexers are controlled by four variables (`S0` through `S3`) which depend on the membrane. To physically propagate the bitstream, a set of three input (`SIC`, `WIC`, and `NIC`) and three output (`SOC`, `EOC`, and `NOC`) lines are also required. Finally, we designated a dedicated, global input line (`GIC`) to handle the *entry points*, that is, the elements in the southwest corner of each block. These elements have to be handled separately because the represent the places where the configuration bitstream first enters each block. If we want the configuration of the blocks to occur in parallel, we therefore need a global line, that is, a line which is can be accessed in every element of the array at the same time.

Careful observation of the programmable network should reveal that a set of 4 communication patterns (i.e., four sets of values for the variables `S0` through `S3`) can describe the propagation path through any block (Fig. 4-5).

The structure of MuxTree does not allow us to determine a precise figure for the time required to completely configure an array of elements. In fact, the speed of the propagation, and thus the time required for the configuration of the FPGA, depends on the clock which controls the register. Whereas in conventional FPGAs the maximum frequency of this *configuration clock* is independent of the application, such is not the case for MuxTree. The maximum clock frequency depends essentially on the longest combinational path the configuration signal has to traverse. In conventional FPGAs, the configuration propagates though a fixed path, and the frequency can thus be determined independently of the configuration itself. In MuxTree, the propagation path for the configuration depends on the size of the blocks, and thus on the application.

We cannot therefore determine *a priori* the maximum frequency of the configuration clock. On the other hand, we can assume that the configuration register will be able to operate at a much greater frequency than the flip-flop used in the element's functional unit. Therefore, we will assume that there exists in our system a configuration clock `CCK`, distinct from the functional clock `FCK`, and operating at a much higher frequency.

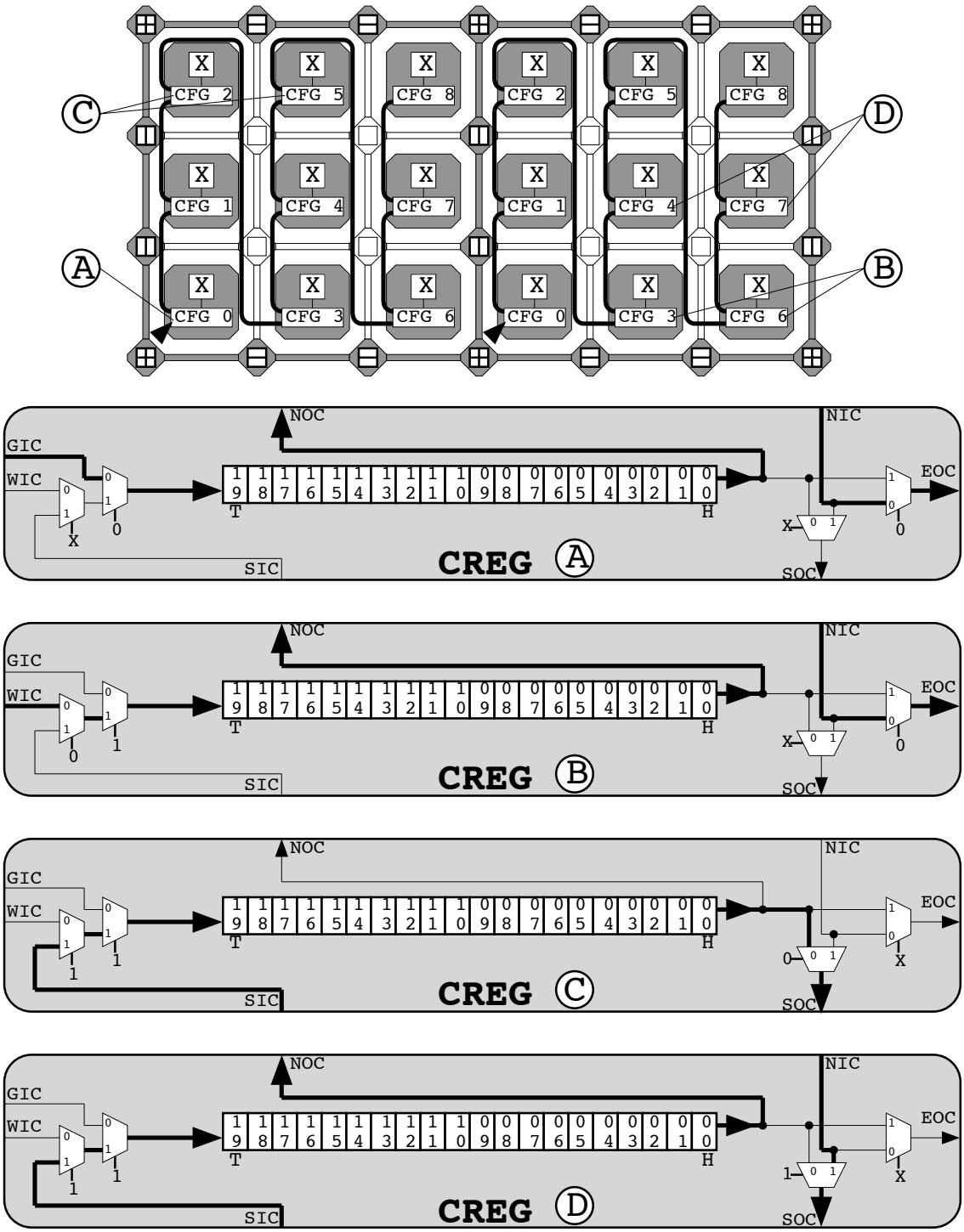

Figure 4-5: The propagation path for the configuration within two 3x3 blocks of elements and the four possible patterns required for the propagation: (A) southwest corner element (point of entry), (B) bottom row, (C) top row, (D) all other elements.

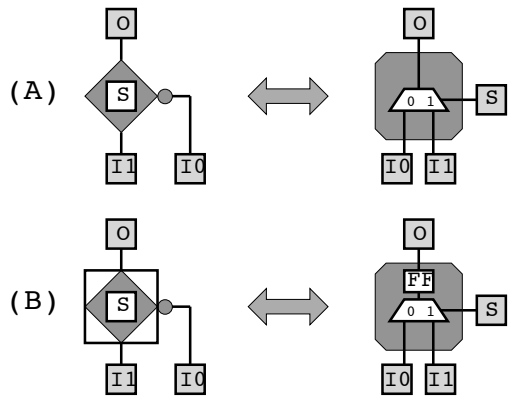

Figure 4-6: Comparison between a BDD test element and a MuxTree element for combinational (A) and sequential (B) behavior.

4.1.4 MuxTree and Binary Decision Diagrams

The architecture of MuxTree, and particularly its programmable function, bears a fairly strong resemblance to that of the soon-to-be-discontinued Xilinx 6200 family of FPGAs[2] [113], also based on a single two-input multiplexer coupled with a flip-flop. The main difference between the two architectures (apart form the configuration mechanism) lies in the connection network: whereas the 6200's network is perfectly symmetric, in MuxTree the connections (and particularly the short-distance network) are strongly directional. This unusual structure is the result of a very careful analysis and is designed to allow an array of MuxTree elements to be easily configured as a *binary decision diagram* (BDD).

A binary decision diagram [4, 16, 57] is essentially a compact representation of a *binary decision tree*, which in turn is a means to describe combinational functions. If the minimization of binary decision diagrams is a complex problem, deriving the binary decision tree for any given combinational function is trivial. In other words, it is very easy to find *a* binary decision diagram which realizes a given function, but it can be very hard to find the *minimal* diagram for a function

Nevertheless, BDDs remain a powerful representation method for combinational functions, and MuxTree was designed to directly implement such diagrams: the functional unit of a MuxTree element is patterned after the basic unit of BDDs, the *test element* (Fig. 4-6). A test element represents a choice: if the test variable `S` is `0`, then the output `O` will be equal to the input `I0`, else (if `S=1`), `O` will be equal to `I1`.

It should be obvious that the behavior of a test element is identical to that of a two-input multiplexer controlled by the variable `S`, that is, identical to that of the functional unit of a MuxTree element. A slight complication is introduced by the need for sequential behavior ("standard" BDDs are exclusively combinational), but the transition from the description of a function, through a BDD representation, to a MuxTree configuration remains relatively effortless[3].

2. Which is not too surprising, since both are based on the now-defunct Algotronix family of FPGAs.

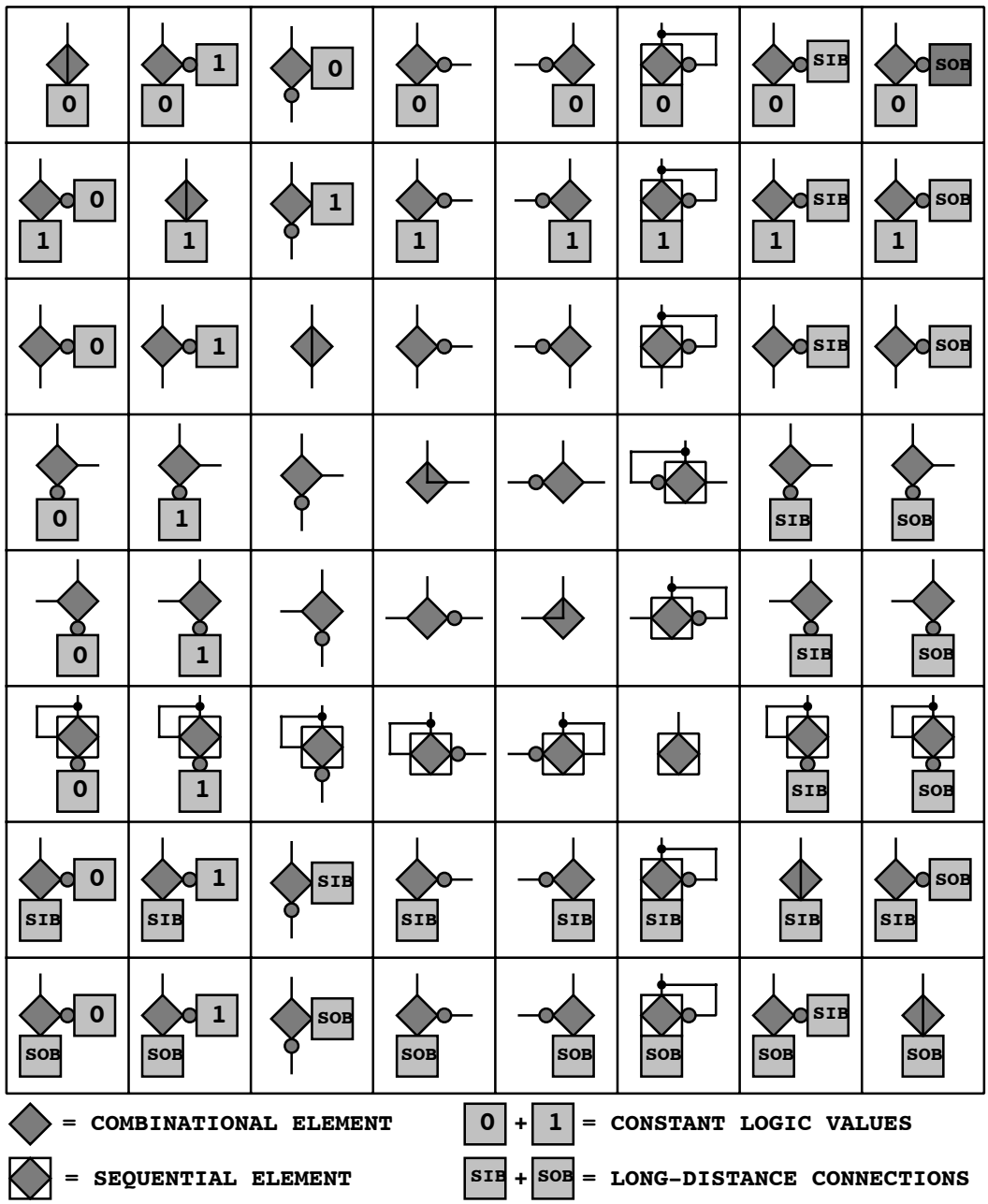

Figure 4-7: Exhaustive table of possible configurations for the programmable function of a MuxTree element seen as BDD nodes.

The desire to model BDD test elements then explains many of the peculiar features of MuxTree. In particular, the unusual pattern of short-distance connections is designed to simplify the task of bringing the inputs to the test element, and the set of programmable inputs to the functional unit was selected to allow it to implement all possible configurations of a BDD test element (Fig. 4-7).

3. Which is a considerable advantage in relation to more "conventional" FPGAs, for which the transition from function to configuration can be extremely complex and time-consuming.

4.2 Self-Test in MuxTree

Any literature search, however superficial, on the subject of testing will reveal the existence of a considerable variety of approaches to implementing self-test in digital circuits [1, 45, 52, 63, 80], including some which can be applied to FPGAs [2, 41, 95, 96, 106]. Even though we exploited to the greatest possible extent the existing knowledge base in the development of our own system, we found that the peculiar requirements of our bio-inspired systems prevented the use of off-the-shelf approaches. In the next subsections, after a brief introduction to the main categories of conventional testing approaches, we will outline the constraints imposed by bio-inspiration, and then proceed to describe in some detail the solutions we adopted to add self-testing capabilities to our FPGA.

4.2.1 Testing Digital Circuits

In order to better understand the testing mechanism we introduced to allow the detection of faults in an array of MuxTree elements, this section will attempt to provide a very cursory introduction to *fault modelization* and *fault classification*, so as to be able to place our approach in the more general framework of digital circuit testing. For a more detailed analysis and classification of testing methods, a number of sources are available [1, 52, 80].

An accurate model for physical faults is a prerequisite to the analysis of any testing method. Unfortunately, such a definition is far from simple, and fault modeling remains a subject of considerable research efforts. Apart from design errors, which, strictly speaking, cannot be considered actual faults (but nevertheless require testing, if only at the prototype stage), physical faults can occur for a number of different causes. Among the more common such causes, we can mention fabrication defects (more and more frequent as fabrication technology improves), electron migration, cosmic radiation (relatively minor on the surface of the earth, but a major cause of concern when designing circuits for space applications), and various environmental factors (heat, humidity, etc.). Moreover, whatever their cause, these faults can have many different effects on the physical substrate of a VLSI circuit, such as increased delays on signals, bridges between lines, or broken connections. To further complicate the design of a testing system, the possibility of multiple faults occurring within a single circuit, as well as the transient or intermittent nature of many physical defects, should be taken into consideration.

In theory, a complete testing system should be able to handle all possible faults in a circuit, whatever their origin or effect. In practice, the sheer number of different possibilities prevents such a complete coverage. The conventional approach to the design of testing systems consists of handling faults from the point of view of their effect on the logic behavior of the circuit. In particular, the most common modelization of physical defects is as *stuck-at* faults: the only defects which are actually considered are those which have the net effect of fixing

the value of a line to 0 (*stuck-at-0* faults) or 1 (*stuck-at-1* faults). This model, which by no means covers all possible electrical faults in a circuit (for example, it does not accurately model physical faults which cause lines to be left floating in a high-impedance state, or faults which cause bridging between lines), is nevertheless the most commonly used in the development of testing algorithms[4].

Because of the wide variety of fault models and of approaches, a complete classification of testing methods is extremely difficult. In the following paragraphs, we will classify testing approaches according to two criteria: off-line vs. on-line testing, and external vs. self-testing. The first classification is based on whether the test occurs while the circuit is operating (on-line test) or in a separate, dedicated phase (off-line test). The second criterion addresses the difference between circuits where the logic required for testing resides within the circuit itself (self-test) and those which rely on external devices (external test). This classification, however incomplete (see [1] for a much more exhaustive effort), will be useful to place our own testing mechanism in the framework of the test of digital circuits.

Off-line external test is by far the most common approach to testing digital circuits. This category includes a wide variety of techniques, applied both at fabrication and on the field. The advantage of testing circuits at fabrication, that is, before they are packaged, is that it is possible to access lines and elements in the interior of a circuit. Once a circuit is packaged, in fact, the only accessible points are the input/output pins of the chip. On the other hand, as long as the entire circuit is accessible, special machinery (e.g., bed-of-nails systems, electron beams) can access any line in the circuit to read or, in some cases, set its logic value.

Whenever it becomes necessary to test the circuit on the field, where only the input/output pins can be accessed, other off-line external testing techniques exist. The most frequently used approach is to apply a set of dedicated input patterns and observe the corresponding outputs of the circuit. These outputs are then compared to a set known to be correct[5], any discrepancy revealing the presence of a fault. The main advantage of such a technique is that, obviously, no additional logic is required inside the circuit, a major bonus for commercial applications. The drawbacks are that this kind of test is usually fairly time consuming (due to the number of input patterns required to test all or most of the lines in the circuit), that it can be difficult to find a set of input-output patterns capable of detecting most or all faults (particularly for sequential circuits), and that can be very hard to identify the exact location of a fault in a circuit.

It is not simple to identify a technique belonging to the category of *on-line external testing*. Such a technique would require that the test occur while the circuit is operating, but using external devices. We are not aware of the existence of such a technique.

4. Of course, alternative methods exist. To name but one, IDDQ testing [80] tries to detect faults by monitoring fluctuations in the circuit's power consumption.

5. The comparison can either be effected on the complete output patterns or on a compressed version (signature).

Off-line self-test is a relatively common solution for circuits which require testing on the field. By integrating additional logic in the circuit itself, it is possible to overcome some of the drawbacks of external testing (for example, by allowing access to some of lines and elements in the interior of the circuit). Among the most common test techniques belonging to this category, we can mention the use of *scan registers*, where some or all the memory elements (flip-flops, registers, etc.) are chained together to form one long shift register, so that it becomes possible to retrieve a "snapshot" of the state of the circuit at any given time.

Another common off-line self-test technique is a variation of the I/O pattern approach used for external testing. In this case, the set of input patterns, rather then being stored outside the chip, is algorithmically generated (usually pseudo-randomly) inside the circuit itself. This kind of approach, while rarely guaranteeing that one hundred percent of faults will be found, is nevertheless able to detect the great majority of faults with a relatively small amount of additional logic. Often, however, most of the additional logic required by this technique is dedicated to the analysis of the output patterns: strictly speaking, self-test requires that the correctness of the outputs be verified within the circuit itself (*self-checking* circuits), a task often more complex than the generation of the inputs.

The considerable amount of additional logic required by all *on-line self-test* techniques has prevented them from gaining a wide acceptance, particularly in the commercial world. Mainstream application do not require the ability to detect faults while the circuit is operating, since it is normally not vital that the correctness of the results be assured. Critical applications usually rely on duplication or triplication, where the outputs of two or three complete circuits are compared to detect faults. This approach, while it guarantees the detection of basically all single faults which can occur in a circuit, is nevertheless somewhat limited in its performance, as it cannot guarantee the detection of multiple faults.

On-line self-test techniques are usually developed anew for each circuit. It is therefore difficult to define a "standard" technique. In data paths, self-test is often implemented using *error-detecting* codes [45, 47]: by using redundant bits to store information, it is possible to detect errors which alter the value of the data (the simplest method to achieve this kind of effect is the use of a single *parity* bit). In general, however, on-line self-test relies on the particular structure and functionality of each circuit to try and exploit distinctive features and properties to verify the behavior of the circuit. This lack of standard approaches is, of course, another reason for the rarity of commercial on-line self-testing circuits.

Thus, because of the relatively important hardware overhead, self-test has, until recently, been considered too expensive for extensive use in mainstream commercial applications. In particular, on-line self-test, while routinely applied in software, is very rarely exploited at the hardware level. Conventionally, circuits are subjected a non-exhaustive off-line test after fabrication so as to eliminate obviously faulty chips (*quality assurance*), and many faults are detected only when the circuit fails to operate correctly, at which point it is replaced. However,

as circuits become more and more complex (implying an increase in the probability of faults being generated during fabrication) and expensive to manufacture, some basic self-test techniques are starting to be integrated in their design and quickly becoming indispensable.

While it is unlikely that on-line self-test will soon become a regular feature for mainstream applications, critical applications (e.g., circuits which operate in space, some embedded circuits such as those used for automotive applications, etc.) are likely to exploit such techniques at least to some degree since, unlike the simple duplication of the circuit, they can allow circuits to operate in the presence of more than one fault with an often smaller amount of additional logic.

This kind of consideration also applies to FPGA circuits. As they are often used for prototyping, commercial FPGAs (e.g., the Xilinx XC4000 series [111], the Altera FLEX series [6], and most others) often include some form of self-test, usually in the form of a *boundary scan* mechanism (a scan register limited to the input/output pins of the chip), meant to aid in the debugging of novel designs. However, as they become more and more complex (FPGAs have the potential of becoming some of the most complex circuits in the market), manufacturers are more and more interested in being able to detect faults not only in the designs being prototyped, but also in the circuits themselves. FPGAs developers are assisted in this task by the regular, modular structure of the arrays, as well as by the fact that they can be reconfigured so as to facilitate the detection of faults. Even further, the regularity of the arrays can also be exploited to achieve self-repair, the natural step beyond self-test.

4.2.2 Constraints

The first step in the development of a self-repair system is thus to endow our FPGA with self-test. In our case, the task is complicated by a number of constraints imposed by the overall approach of Embryonics.

The first added constraint is imposed by the need for self-repair: any self-repair mechanism will be able to know not only that a fault has occurred somewhere in the circuit, but also exactly where. Thus, our system has to be able to perform not only *fault detection*, but also *fault location*. Moreover, our self-repair mechanism, which, as we will see, uses spare elements to replace faulty ones and reroutes the connections accordingly, implies that certain kinds of fault (for example, faults on certain connections) are not *repairable*, as they prevent the reconfiguration of the array.

A second constraint is that we desire our system to be completely distributed. As a consequence, the self-test logic must be distributed among the MuxTree elements, and the use of a centralized system (fairly common in self-test systems) is not a viable option. This constraint is due to our approach to reconfiguration: by assuming that any element can be replaced by any other (an assumption which, as we will see, is a direct consequence of our self-replication mechanism), we need our array to be completely homogeneous.

The relatively small size of the MuxTree elements also imposed a constraint on the amount of logic allowed for self-testing. While the minimization of the logic was not really our main goal, and we did indeed trade logic in favor of satisfying our other constraints, we nevertheless made an attempt to keep the testing logic down to a reasonable size, which imposed some rather drastic limitations on the choices available for the design of our mechanism.

In addition to these constraints, more or less imposed by the features of our system, we also decided to attempt to design a self-test mechanism capable of operating on-line, that is, while the system is executing the application and transparently to the user. While biological inspiration was not, in this case, the main motivation behind our choice, such an approach is entirely in accordance with the behavior of biological systems: organisms monitor themselves constantly, in parallel with their other activities. This self-imposed constraint is extremely strong: to obtain an on-line self-test system while at the same time minimizing the amount of additional logic is a nearly impossible task. As a consequence, in our actual implementation this constraint was somewhat relaxed, and we will see that our self-test will occur on-line only partially.

In conclusion, we tried our best to respect the many external constraints, some of which contradictory, when designing our self-test mechanism. These constraints, and in particular the need for a completely distributed on-line mechanism, excluded the use of most of the "standard" approaches to self-testing (output pattern analysis, parity checking, redundant coding, etc.). The result is a system which falls somewhat short of our ideal requirements, but is nevertheless powerful enough for our purposes.

4.2.3 The Programmable Function

Thanks to the experience we acquired in the design of an integrated circuit containing an array of MuxTree elements (without self-replication or self-repair), we were able to determine that the functional part occupies approximately 10% of the total silicon area of a single element. Taking into consideration its relatively small size, we decided to test the functionality through complete duplication. Obviously, the presence of two identical copies of the sub-circuit allows to detect faults on-line through a simple comparison of their outputs (Fig. 4-8).

If our final objective was limited to the detection of faults, comparing the two outputs would be sufficient. However, a certain number of complications are introduced by the need for self-repair. As we will see in section 4.3, one of the requirements of self-repair is that the circuit be able to resume operation after being repaired without losing the information it stored at the instant the fault was detected. For an element configured as purely combinational, this requirement has no direct consequence: replacing the faulty element with a faultless one is sufficient to correctly repair the circuit. However, if the element's configuration requires the use of the internal flip-flop, it becomes necessary to keep faults from causing the incorrect value to be stored, which would corrupt the information

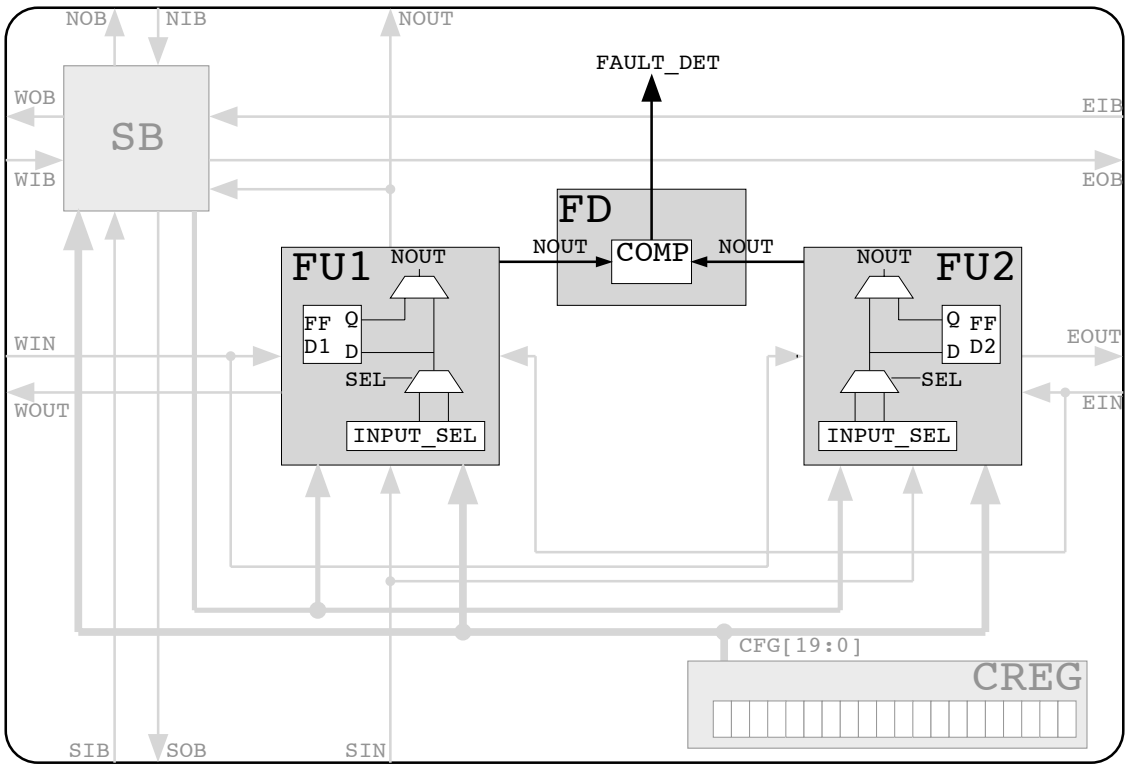

Figure 4-8: Comparing the outputs of the two functional units allows the detection of faults.

held in the circuit and prevent the resumption of operation. To prevent the wrong value from being memorized, we must therefore introduce a further comparison between the *inputs* of the two flip-flops (Fig. 4-9).

The desire to implement self-repair imposes a further addition to the self-test system. In fact, while two copies of the internal flip-flop are sufficient to detect the presence of a fault *within* the flip-flop itself, they cannot guarantee that the correct value will be used when repairing the circuit. Obviously, a difference between the outputs of the two flip-flops testifies to the presence of a fault (since their two inputs were tested, the fault must reside within one of the memory elements), but can not identify which one is faulty. In order to ensure that the repair mechanism will not access the wrong value, we were forced to introduce a third copy of the flip-flop (Fig. 4-10). A simple 2-out-of-3 majority function will then suffice to ensure that self-repair will indeed preserve the correct value.

4.2.4 The Programmable Connections

MuxTree is a relatively connection-intensive circuit. In fact, while the network joining the elements is not very complex, the small size of the functional part implies that a fault is as likely to occur in a connection as in the logic gates which implement the element's functionality.

An immediate solution to the problem of testing the connection network is to proceed as we did for the functional part, by using duplication. While not exclud-

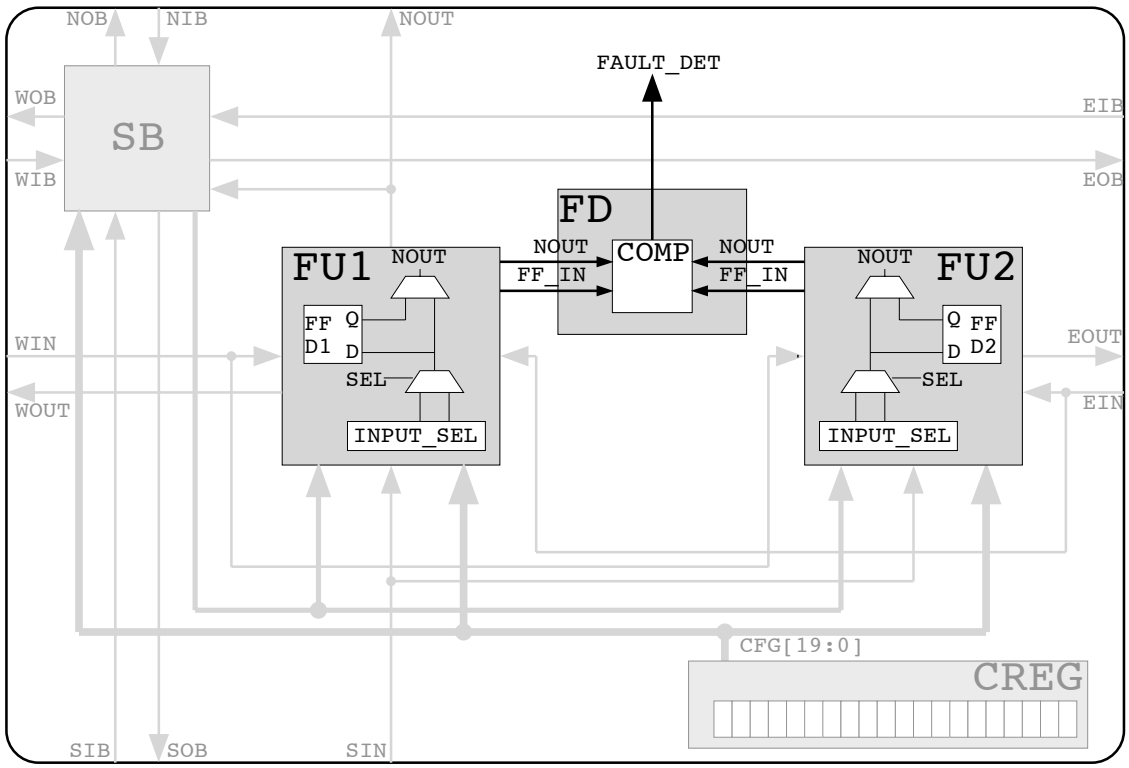

Figure 4-9: Comparing the inputs of the flip-flops prevent the wrong value from being memorized.

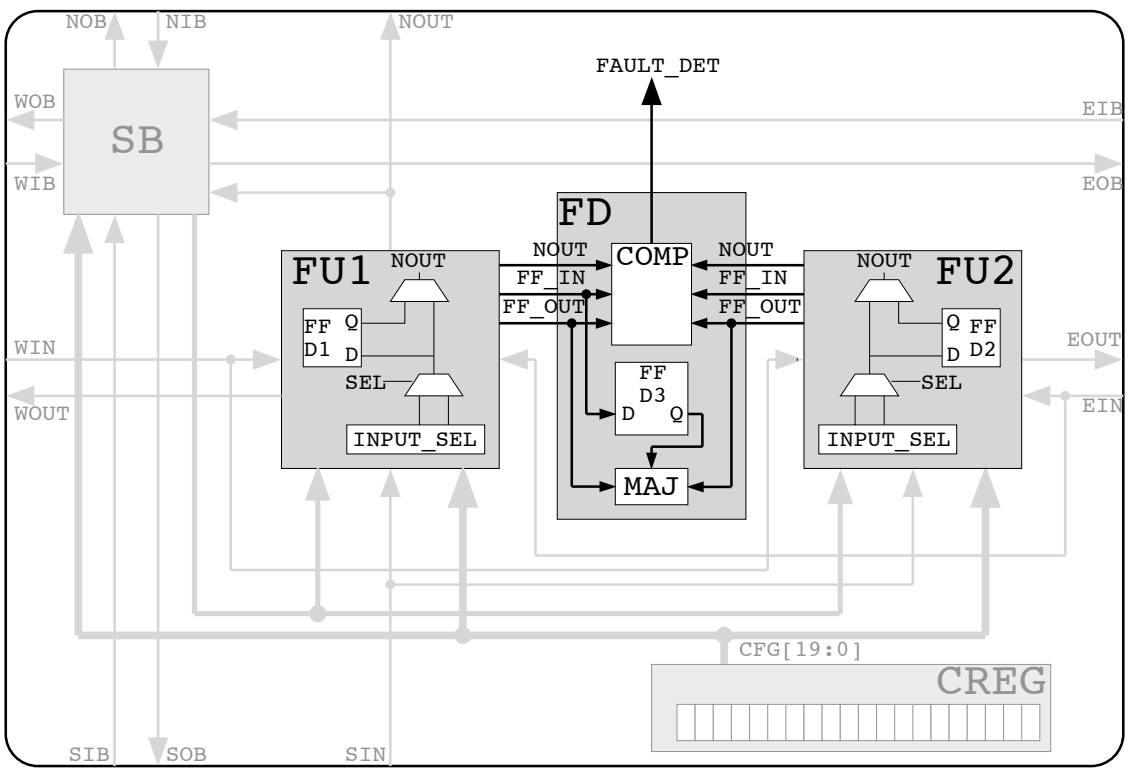

Figure 4-10: A third flip-flop and a 2-out-of-3 majority function allow us to recover the correct value even if a flip-flop is faulty.

ing the possibility of implementing either a partial or a full duplication[6], a practical observation associated with self-repair induced us to decide not to test the connection network. In fact, the only practical way to achieve rerouting in a simple network such as MuxTree's is to exploit the existing set of connections, a strategy which would not be possible without the assumption that the lines are indeed faultless. In other words, faults in the connection network might very well be detectable, but are not likely to be repairable.

Of course, we did not ignore the possibility of exploiting techniques other than duplication for testing the connection network. For example, we explored the possibility of reserving a certain portion of the clock period for testing: by propagating predetermined values through the network, it would be possible to detect faulty lines. This solution, which would probably impose a smaller (but still relatively important) hardware overhead compared to duplication (at the expense, of course, of an increase in the duration of a clock period), nevertheless still fails to address the question of repairability.

In conclusion, none of the mechanisms we examined was found to meet all our constraints, either because they imposed a considerable hardware overhead or because of unavoidable conflicts with the self-repair mechanism. As a consequence, we decided to temporarily desist in our attempts to design a self-test mechanism for the connections, while remaining fully aware of its importance and ready to investigate possible solutions.

4.2.5 The Configuration Register

Testing the configuration register poses a similar set of problems, but its relatively large size (an estimated 80% of the surface of an element) makes it imperative that some form of testing be applied.

As we mentioned, the constraints for the self-test of the configuration register are the same as for the rest of the element: the self-test mechanism must require a limited amount of additional logic, should ideally be on-line and transparent, and should be compatible with self-repair.

As was the case for the connection network, it is this last requirement which proved most restrictive. In fact, reconfiguring the array implies that a spare element will need to assume the functionality of the faulty one, and therefore its configuration. Since each element contains a single copy of the configuration register (its size precludes the possibility of duplication), a fault which modifies the values stored in the register results in a loss of information, which cannot easily be recovered[7]. In addition, most of the faults which could affect the configuration

6. Partial duplication would involve having two copies of the switch block, but a single set of connections, while full duplication would add a second, disjoint communication network (which, incidentally, would complicate fault location).

7. We considered the possibility of encoding the configuration using some form of self-correcting code (i.e., a redundant encoding which allows the correct information to be retrievable in the presence of a given number of faults), but after examining several options we concluded that all such codes would require an hardware overhead we deemed unacceptable.

register would prevent its operation as a shift register, thus requiring a separate mechanism to transfer the faulty element's configuration to the spare element. Such a mechanism, while in theory not excessively complex, would nevertheless require an unwarranted amount of additional logic.

An interesting observation which can have an impact on the development of a self-test mechanism for the register is that the configuration of an element is not supposed to change once the circuit has been programmed. While this observation does not address the question of repairability, it might simplify the task of fault detection: any change in the value stored in the register during normal operation must necessarily be the consequence of a fault.

Unfortunately, detecting such a change, while somewhat simpler than other detection methods, still requires too much additional logic to be viable[8]. We feel that the hardware overhead could be vastly reduced by exploiting transistor-level design techniques, and plan to more closely investigate such approaches when we will have the opportunity to implement them[9].

So once again we were not able to find an approach which would satisfy all our requirements. Nevertheless, the probability of a fault occurring in the register being greater than for any other part of the element, we decided that some degree of testing was a necessity. Examining our system, we determined that relaxing the requirement that the self-test occur on-line would allow us to design an extremely simple mechanism.

Our approach is based on the observation that, when the circuit is programmed, the configuration bitstream is shifted into the register. It would therefore be very simple to include in the bitstream a *header*, in the form of a dedicated testing configuration. This header, which will be shifted into the register of all the elements in parallel, will not correspond to any particular functionality, but will rather be a pattern designed to exhaustively test that the configuration register is operating correctly. By using all registers in parallel we can make the length of the pattern independent of the number of the elements in the system, and by entering the pattern *ahead* of the actual configuration we can avoid the loss of information.

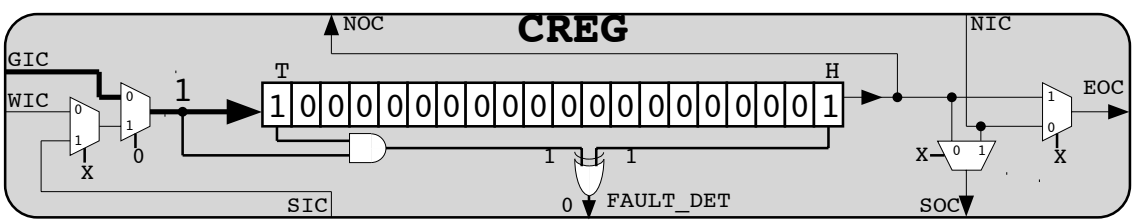

Figure 4-11: The test pattern `180001` which allows faults in the register to be detected.

8. A straightforward implementation would require N-1 `XOR` gates to detect single faults in a N-bit register.

9. At the moment, we do not have access to a foundry: all the prototypes we developed exploited commercial FPGAs, and were therefore limited to gate-level design.

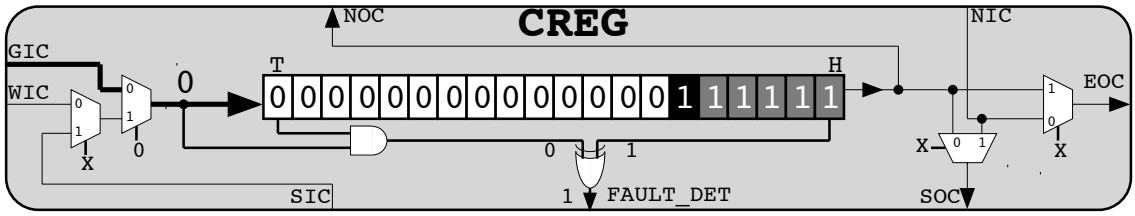

Figure 4-13: A stuck-at-1 fault corrupts the test pattern and is therefore detected.

A pattern which meets our requirements (together with the gates required to detect the faulty operation of the register) is shown in Fig. 4-11. Its length corresponds to the length of the register to be tested plus one (the value of the input line used to propagate the pattern), and it is able to detect any defect in the operation of the shift register. The pattern (`180001` in hexadecimal notation) enters the configuration register as a "normal" configuration, i.e., it is shifted in from left to right. If no fault is present, the pattern's leading (right-most) `1` reaches the head `H` of the register at the same time as the trailing `11` comes to occupy its tail `T`. The `AND` gate will therefore output `1` at the same time as the head of the register becomes `1`, and the inputs of the `XOR` gate, from `00`, will become `11`. Thus, at no moment does the output of the `XOR` gate become `1`, and thus no fault is detected.

Let us for example assume that a stuck-at-0 fault is forcing the output of one of the register's elements to logic value `0` (Fig. 4-12). The leading `1` of the pattern will enter the shift register from the left, and start to travel through the register from left to right. When it encounters the stuck-at-0 fault, it will "vanish", and thus never reach the head of the register. When the trailing `11` reaches the register, the `AND` gate will (for the first time) output a `1`. The inputs of the `XOR` gate will then be `10`, which will allow the system to detect the fault. Any stuck-at-1 fault will be similarly detected: the string of `1`s generated by the fault will reach the head of the register before the trailing `11` of the test sequence can arrive at the tail (Fig. 4-13).

This very simple pattern is therefore able to detect a fault anywhere in the register, with the single exception of a stuck-at-0 fault in the very first (left-most) memory element: it should be obvious that if the first element remains stuck to logic value `0`, not only will the pattern's leading `1` never reach the head of the register, but the output of the `AND` gate will equally be stuck at `0`, thus never triggering the detection mechanism.

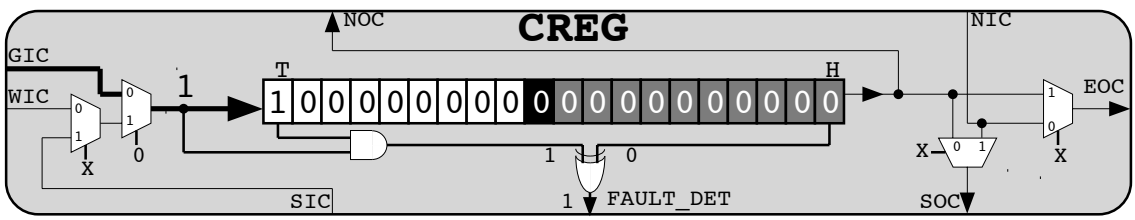

Figure 4-12: A stuck-at-0 fault corrupts the test pattern and is therefore detected.

In practice, however, there is no need for an additional mechanism to resolve this issue, as we can exploit existing material: by "chaining" the three flip-flops in the functional unit of the element to the shift register we can use the existing majority function to test the tail of the register (Fig. 4-14). This mechanism, which might appear exceedingly complex compared to the benefits it provides (the probability of a fault occurring in the last bit of the configuration register is, after all, relatively small), is in reality a requirement for self-repair. In fact, we introduced the third copy of the flip-flop in order to be able to transfer its value to the spare element, and chaining the flip-flops to the register is the most efficient approach to effect this transfer. The mechanism is therefore already in place and can be used for testing at no additional cost.

In conclusion, by relaxing the requirement that the detection occur on-line, we were able to design an extremely simple fault detection system (the hardware overhead consists of only two logic gates) which, as we will see, is perfectly compatible with self-repair.

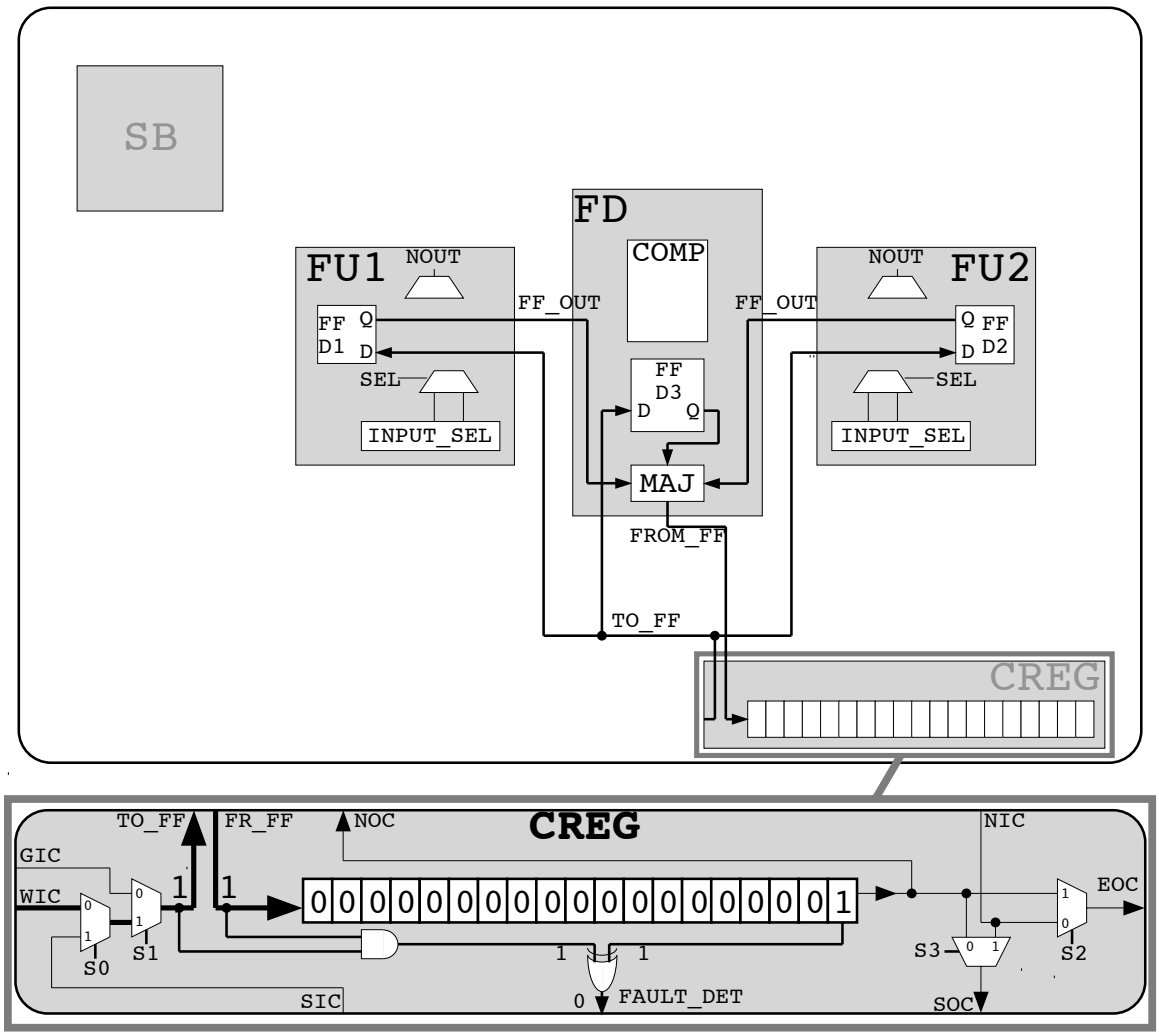

Figure 4-14: The three flip-flops are attached to the configuration register and the register test logic is revised to include the new path.

4.2.6 MuxTree and MicTree

Compared to self-replication (which we described in the previous chapter) and self-repair (which we will describe in the next section), self-test is undoubtedly the area where our implementation has fallen shortest of our goals. While we are aware that our constraints were extremely rigid (maybe too rigid to be strictly realistic), we are nevertheless aware of the weaknesses of our self-test system.

In order to partially compensate for the limitations of self-test at the molecular (MuxTree) level, we plan to introduce further self-testing features at the cellular (MicTree) level. Since our ultimate goal is to use MuxTree as a programmable platform to implement arrays of MicTree cells, we hope that by combining the self-testing capabilities of the two levels we might be able to eliminate some of the weaknesses of each separate mechanism.

Self-test at the cellular level must operate under very different constraints. For example, since we are working with elements which are much more complex, we are not quite as limited in the amount of additional hardware we can introduce for testing, which in turn means that many approaches we had to discard because they were too complex could perhaps be applied. On the other hand, since the cellular level accesses the hardware only indirectly (through the configuration), certain kinds of faults are probably harder to detect, and self-test is unlikely to occur completely on-line.

Nevertheless, by integrating the self-test systems of the two levels, we hope to ameliorate the weaknesses of self-test at the molecular level. To this end, we have already experimented with a possible solution to the testing of a MicTree cell. This solution, conceptually similar to the method presented in [2, 95, 96], involves using half of the cells to test the other half, and then inverting the process. The test is effected fairly simply by adding a dedicated subprogram to the genome, accessed regularly by all the cells at the same time. While not strictly speaking an on-line self-test approach, this solution allows for transparent testing (the subprogram is invariant and can be added to any and all programs) with a limited amount of additional hardware, and is a valid complement to the molecular-level mechanism.

4.3 Self-Repair in MuxTree

As was the case for self-test, there exist a number of well-known approaches to implementing self-repair in two-dimensional arrays of identical elements in general [13, 19, 23, 56, 69, 70, 73, 102, 105] and FPGAs in particular [25, 35, 40, 51, 79, 85]. Most, if not all, of these are variations on a few basic approaches, and rely on two main mechanisms: *redundancy* and *reconfiguration*. As we will see in the next subsections, the system we developed to implement self-repair in MuxTree is no exception, even if it had to satisfy a set of relatively non-standard constraints imposed by the peculiar features of our FPGA.

4.3.1 Repairing Digital Circuits

Self-repairing systems are a particular subset of a more general category of systems, usually referred to as *fault-tolerant* systems, a term which implies the capability of operating correctly in the presence of one or more faults. In particular, self-repair implies that the logic containing the fault be somehow removed from the active part of the system, its functionality being taken over by *redundant*, or *spare*, logic.

In commercial, off-the-shelf systems, fault tolerance is but rarely implemented through self-repair. For example, critical application usually rely on triplication: three copies of the complete system or circuit operate in parallel and their results are input to a *2-out-of-3 majority* circuit which assures that the output will be correct even if one of the copies is faulty. Obviously this solution is considered a final resort, both because it is very expensive and because it allows for a relatively small degree of fault tolerance (two faults on two different copies of the system will invalidate the approach).

A second common approach to fault tolerance is the use of *error-correcting* codes [45, 47]. By using a redundant coding of information, it is possible not only to determine that a fault is present within a word (as was the case for the error-detecting codes mentioned in subsection 4.2.1), but also to recover the correct value. This increased versatility is achieved though the use of additional redundant bits, and is consequently more expensive in terms of additional logic. As for error-detecting codes, this technique is also ill-suited for fine-grained FPGAs, as it requires multiple-bit words.

Self-repair is a technique which achieves fault tolerance with a somewhat different approach. Rather than extracting the correct result from a faulty but redundant output, it aims at producing the correct output by removing the fault from the circuit. Since current technology does not allow the fault to be removed physically, self-repair relies on a reconfiguration of the circuit which reroutes the signals so as to avoid the faulty areas. This technique, while often capable of achieving considerable fault tolerance with a relatively small amount of additional logic, is obviously more complex to implement, as it requires both the ability of identifying the exact location of a fault in the circuit and the presence of redundant logic capable of replacing the functionality of the faulty part of the circuit. These requirements have the effect of limiting the practical use of self-repair to arrays of identical elements, where a faulty element it can be replaced by a spare which, being identical, can take over its functionality.

The amount of additional logic required by this approach can vary widely, depending not only on the number of the spare elements, but also on their placement within the array. In fact, the position of the spare elements, which can vary considerably from one implementation to the next, affects the complexity of the logic necessary to reroute the signals around the faulty elements. In practice, this kind of reconfiguration schemes are usually applied to arrays of complex processors, where the probability of faults occurring in the system justifies the rela-

tively complex rerouting logic. Obviously, the regular structure of FPGAs is perfectly suited to this kind of approach, even if the small size of an FPGA element requires an important effort in minimizing the amount of rerouting logic, thus preventing the use of complex reconfiguration schemes. Nevertheless, reconfiguration is clearly the most suitable approach to introducing fault tolerance in an FPGA circuit.

As we mentioned, commercial fault-tolerant systems are exceedingly rare: systems capable of operating on the presence of faults imply in all cases a relatively important amount of redundancy, which is often considered wasteful for non-critical applications where VLSI chips or, more usually, circuit boards can be replaced without causing excessive problems for the user. In general, fault tolerance has not been considered a high enough priority to warrant the additional cost, with the possible exception of very large distributed systems where the sheer amount of circuitry involved considerably increases the probability of faults.

Introducing fault tolerance in single VLSI chips in general, and FPGAs in particular, has therefore been considered too expensive to be commercially viable. This outlook, however, appears to be changing. As circuits in generals, and FPGAs in particular, become more complex, their yield (i.e., the percentage of chips which exit the fabrication process without faults) is decreasing, and chip manufacturers are becoming more and more interested in being able to produce chips capable of operating in the presence of faults. FPGAs are ideally suited for this kind of approach, as their regular structure allows for the possibility of reconfiguration, and some of the leading manufacturers are beginning to seriously consider the option of introducing fault tolerance in their circuits [7, 8, 31, 85].

4.3.2 Constraints

A mechanism which allows an electronic circuit to be repaired need obviously be very different from that exploited by nature in biological organisms. Since we are not able, given the current state of the art, to physically repair a faulty MuxTree element, we must necessarily provide a reserve of such elements to replace faulty ones (redundancy). Moreover, if we assume that such a reserve is indeed available, we also need to provide a mechanism to allow these spare elements to replace the faulty ones, that is, we need a mechanism to reroute the connections between the elements (reconfiguration).

The extremely small size of our FPGA elements imposes a first set of very rigid constraints. As was the case for self-test, there are serious limitations to the amount of additional logic we can introduce to implement a self-repair mechanism. As we will see, this limitation has serious implications for the choice of possible reconfiguration schemes.

Our self-test system, in order to conserve the analogy with biological organisms, had to operate on-line. For self-repair, this restriction can be somewhat

relaxed. Our self-repair can thus occur off-line, that is, cause a temporary interruption of the operation of the circuit (of course, the process should be as fast as possible). However, it is fundamental that the *state* of the circuit (that is, the contents of all its memory elements) be preserved through the self-repair process, so as to allow normal operation to resume after the reconfiguration is complete.

In practice, the only memory elements in our system are the configuration register and the flip-flop inside the functional part of the element. Since our self-test system allows us to test the register only during configuration, we can also limit its repair to the configuration phase. The assumption that the register is fault-free during normal operation is extremely useful for the reconfiguration of the array, and effectively reduces the requirement that the state of the circuit be preserved to the need to prevent the loss of the value stored in the flip-flops.

The requirement that the FPGA be perfectly homogeneous, which already had a serious impact on the choice of a self-test mechanism, also poses important restrictions on self-repair. Most, if not all, of the existing solution to the problem of repairing two-dimensional arrays rely on some form of centralized control to direct the reconfiguration of the elements. The need for homogeneity severely limits the reconfiguration algorithms which can effectively be adopted in our FPGA.

A final consideration on the desirable features of our self-repair system stems from an analysis of our self-replication process. As we have seen, the self-replication mechanism relies on the construction of an array of identical blocks of elements. It would obviously be desirable, if not strictly necessary, that the self-repair process also be contained inside each block. In other words, we would prefer that any reconfiguration occur within a single block of elements (and thus a single cell). This requirement is far from trivial, since it implies that the location of the spare elements be a function of the size of the block, which is programmable and thus unknown *a priori*. We will see that, by exploiting existing material (notably, the cellular automaton used to define the cellular membrane), we are able to fulfill this requirement with a remarkably small amount of additional logic.

4.3.3 Self-Replication Revisited

In order to design a self-repair system able to operate inside a single block of elements implies that a set of spare elements be included within the block. Since the size of the blocks is programmable, this requirement in turn implies that we cannot designate a set of spare elements before we partition the array.

On the basis of this observation, we can postulate that an efficient approach to designating a set of elements as spares would be to modify the self-replication mechanism to include such a feature. In order to introduce this feature, we modified our cellular automaton. The new automaton (Fig. 4-15) is capable, at the cost of a relatively minor increase in hardware complexity, of performing two novel tasks:

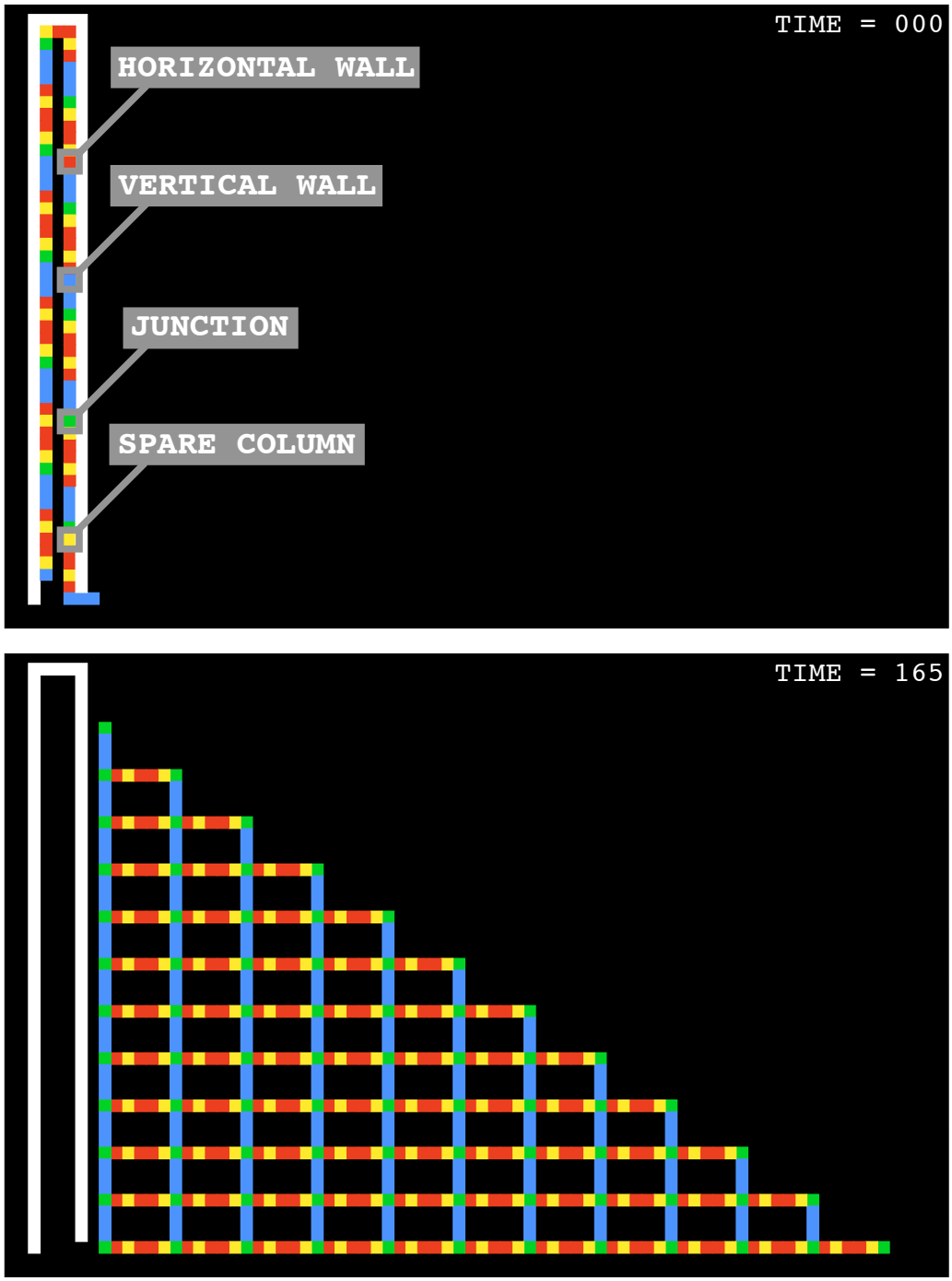

Figure 4-15: The new membrane-building automaton at time 0 (A) and after the sequence had finished propagating (B).

- Partitioning the array into *rectangular* blocks. The old automaton was limited to square blocks, which often represented a considerable waste of resources and complicated the task of generating the configuration bitstream.
- Designating any column as being spare, which allows us to limit our reconfiguration to the interior of a single block.

Without examining the operation of the automaton in detail (its operation is very similar to that of the old automaton, and its hardware implementation will be described in Appendix B), we will simply mention that it uses four different states to describe a membrane:

- a *vertical wall* state (blue), used to delimit the left and right boundaries of the blocks;
- a *horizontal wall* state (red), used to delimit the top and bottom boundaries of the blocks;
- a *junction* state (green), which designate the locations where vertical and horizontal walls meet (i.e., the corners of the blocks);
- a *spare column* state (yellow), which designates a column as spare, and at the same time defines a horizontal boundary.

For example, the particular automaton of Fig. 4-15 would define a block of 6x4 MuxTree elements, with one spare column for every two active ones.

Using the cellular automaton to define which columns will be used as spares is an extremely powerful concept. In fact, it allows us not only to limit reconfiguration to the interior of a block, but also to program the robustness of the system: by adding or removing spare column states to or from the sequence, we are able to modify the frequency of spare columns, and thus the capability for self-repair of the system. Without altering the configuration bitstream of the MuxTree elements, we could introduce varying degrees of robustness, from zero fault tolerance (no spare columns) to 100% redundancy (one spare column for each active column).

4.3.4 The Reconfiguration Mechanism

The new automaton thus allows us to define a set of spare columns within each block. In order to take advantage of these spare elements, we need to develop a system to allow the information stored in a faulty element to be somehow transferred to a working one. As we have mentioned, the information which must be preserved consists essentially of the faulty element's configuration plus the value stored in its flip-flops.

The mechanism we propose to repair faults relies on the reconfiguration of the network through the replacement of the faulty element by its right-hand neighbor (Fig. 4-16). The configuration of the neighbor will itself be shifted to the right, and so on until a spare element is reached.

The amount of logic required by this mechanism is not very important, particularly because, by limiting the reconfiguration (that is, the shift of the configuration registers) to a single direction, we can use the same set of connections used in the configuration of the array. By attaching the contents of the flip-flops[10] to the tail end of the register, we can use the same mechanism to also transfer the contents of the flip-flops, thus completing the reconfiguration.

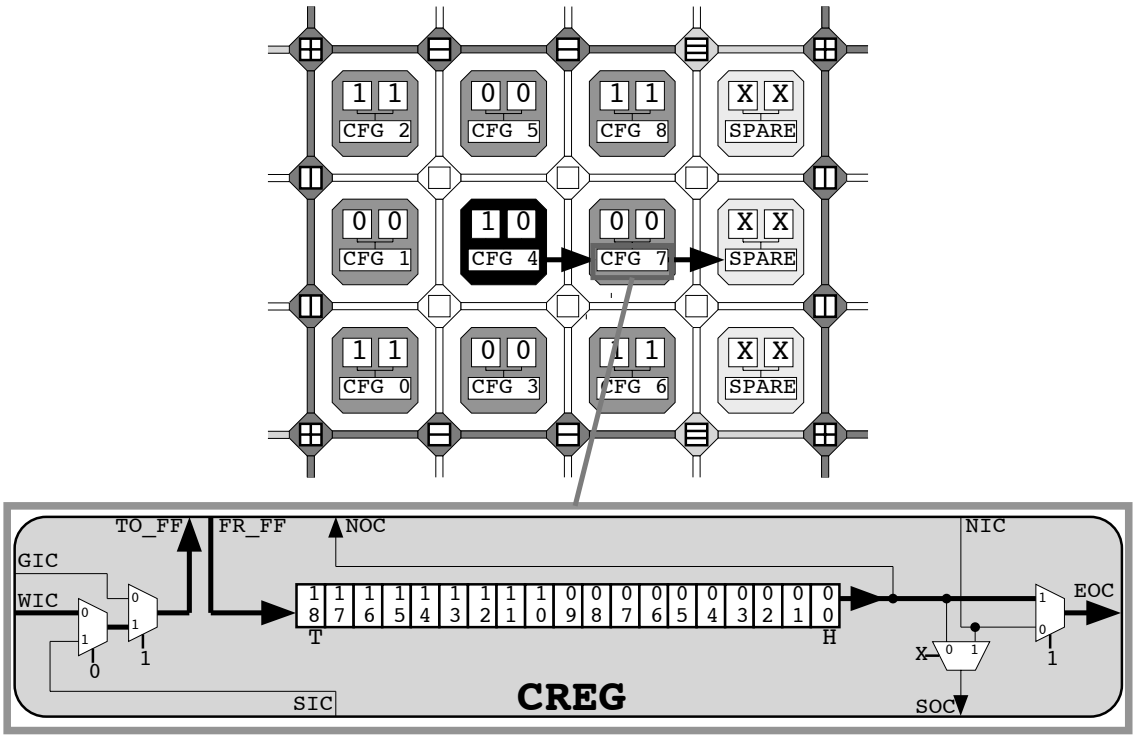

Figure 4-16: A fault is detected in one of the elements and the repair mechanism is activated, causing a shift of the configuration registers.

Once the shift is completed, the element “dies” with respect to the network, that is, the connections are rerouted to avoid the faulty element, an operation which can be effected very simply by deviating the north-south connections to the right and by rendering the element transparent with respect to the east-west connections. The array, thus reconfigured and rerouted, can then resume executing the application from the same state it held when the fault was detected (Fig. 4-17).

Whenever a fault is detected, the FPGA effectively goes off-line for the time required for the element to be replaced (somewhat like an organism becomes incapacitated during an illness). While this inconvenience is acceptable from the point of view of biological inspiration, it is obvious that the interruption in the operation of the circuit should be kept as short as possible, so as to minimize possible data loss. Fortunately, the reconfiguration, being an entirely local mechanism, can exploit the faster configuration clock[11], and thus limit the duration of the interruption.

10. Or, more precisely, the output of the 2-out-of-3 majority function (Fig. 4-14). Note that for purely combinational configuration, shifting the value of the (unused) flip-flops is not strictly required. However, since disabling this feature would actually involve additional hardware, it is more efficient to simply let the mechanism assume that all flip-flops are in use.

11. The reconfiguration requires approximately 20 cycles of the configuration clock. If the difference between the frequency of the configuration clock and that of the functional clock is substantial, it is possible, and even probable, that the reconfiguration will occur transparently to the application.

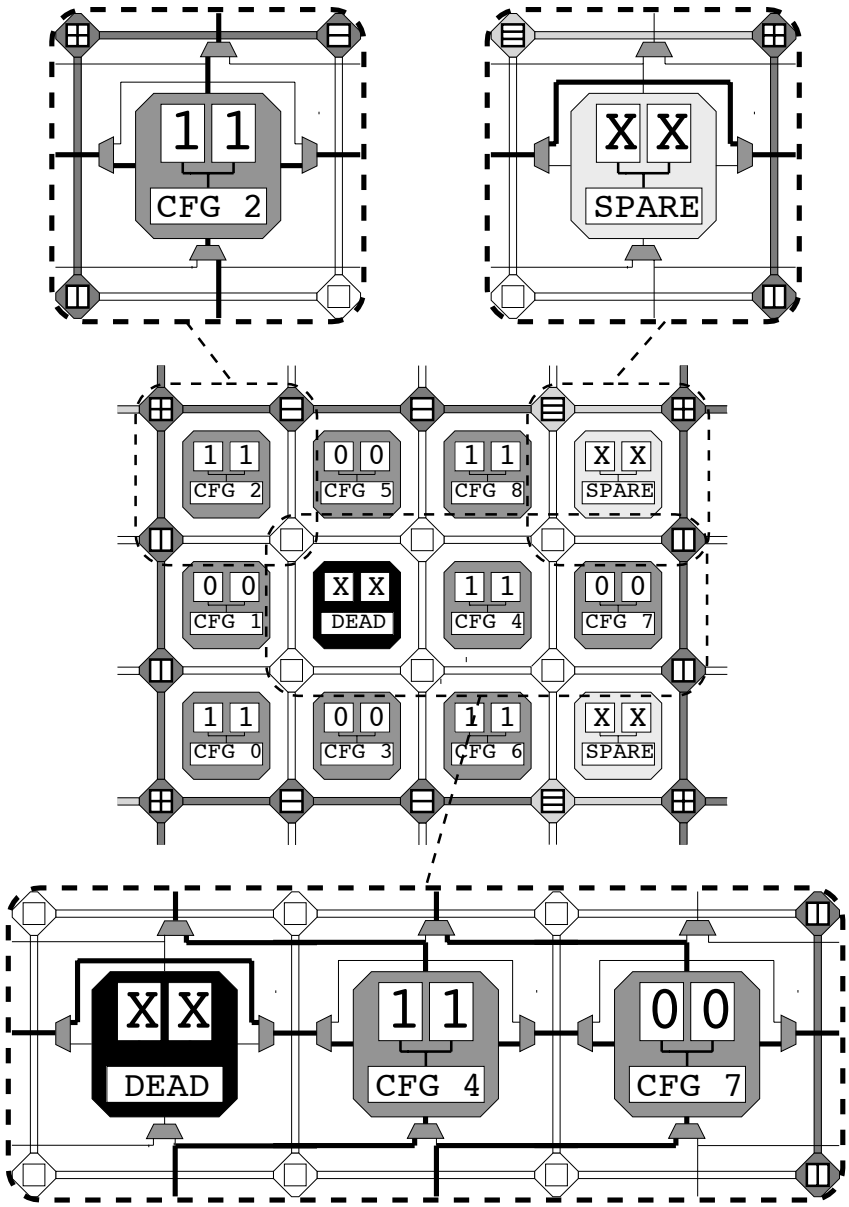

Figure 4-17: The faulty element has been replaced and the circuit can resume operation.

The reconfiguration of the elements through self-repair implies that all connections need to be rerouted to avoid the faulty element. Clearly, such an operation requires a relatively important amount of additional logic in the form of a set of multiplexers which select alternative connecting paths (Fig. 4-17). In order to minimize such additional hardware, we opted for limiting the reconfiguration to a single column. In other words, we do not allow the configuration of an element to be shifted more than once, which has the consequence of restricting the number of repairable faults to one per row between two spare columns. This limitation, while of course not trivial, is however alleviated by the fact that, as we have seen, the frequency of the spare columns can be programmed. Of course, all faults which do not conflict with this restriction can be repaired, leaving a considerable overall robustness to the system (Fig. 4-18).

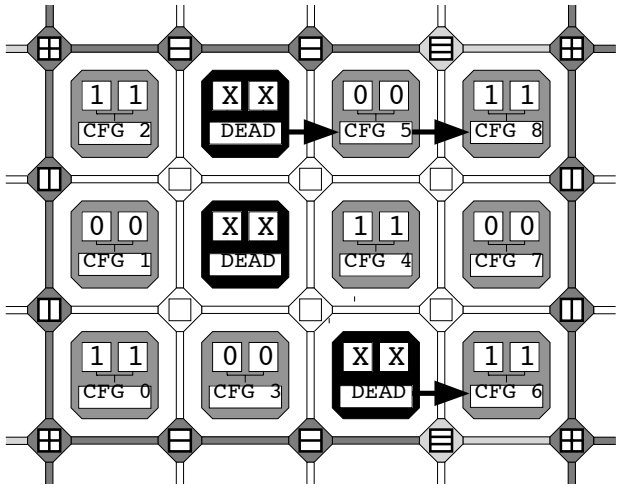

Figure 4-18: As long as the appropriate spare elements are available, more than one fault can be repaired within each block.

The last relevant feature of our self-repair mechanism that we will mention is one that would apparently seem a disadvantage, but reveals itself to be a very positive feature to a more careful examination. We are talking about the fact that the entire self-test and self-repair mechanism we have described is volatile: powering down the circuit results in the loss of all information regarding the condition of the circuit. As we said, this might seem a disadvantage, since it forces us to find and repair the same faults every time the circuit is reconfigured. However, it is crucial to consider that by far the largest part of the faults occurring within a circuit throughout its life are *temporary faults*, that is, faults which will "vanish" very shortly after having appeared [23, 79]. It would therefore be very wasteful to permanently repair faults which are in fact only temporary, and the volatile nature of our mechanism turns out to be an important advantage.

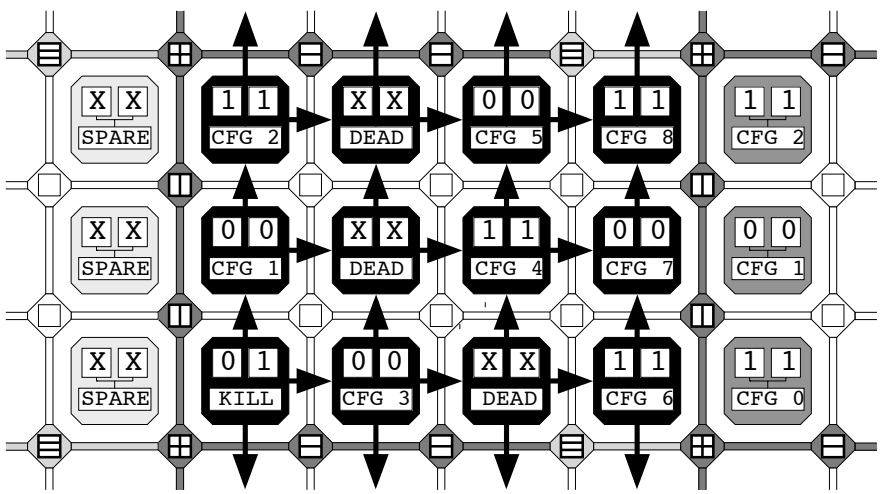

Figure 4-19: The saturation of the molecular-level repair mechanism causes a column of blocks to be "killed".

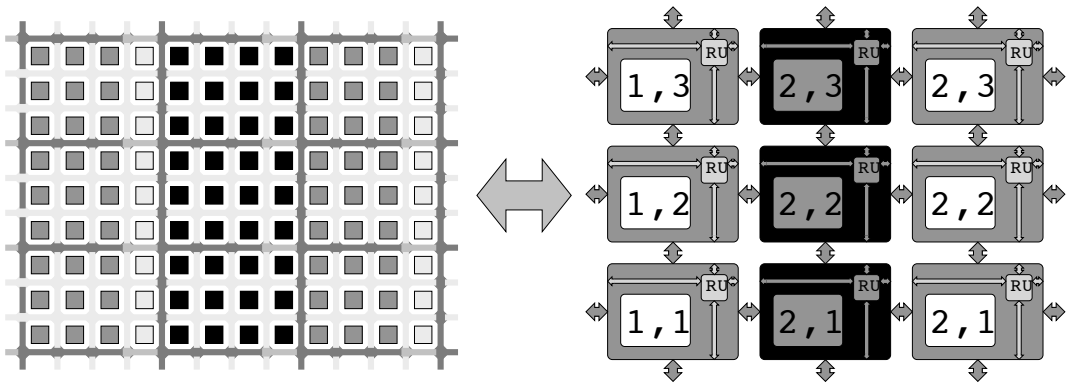

Figure 4-20: The coordinates of the cellular array are recomputed to allow for the destruction of a column of blocks at the molecular level.

4.3.5 MuxTree and MicTree

However versatile MuxTree's self-repair system might be, it is still subject to failure, either because of saturation (if all spare elements are exhausted) or because a non-repairable fault is detected. In order to address this possibility, we implemented a mechanism which, should such a failure be detected, generates a KILL signal which is propagated through an entire column of blocks (Fig. 4-19).

The choice to kill a column of blocks, which might seem arbitrary, is a consequence of the two-layer structure of our system: since a block is ultimately meant to contain one of our artificial cells, killing a column of blocks is equivalent to deactivating a column of cells (Fig. 4-20). At the cellular level, this event will trigger a recomputation of the coordinates of all cells in the system, that is, will activate the cellular-level reconfiguration mechanism. In other words, the robustness of the system is not based on a single self-repair mechanism, which might fail under certain conditions, but rather on two separate mechanisms which cooperate to prevent a fault from causing a catastrophic failure of the entire system.

4.4 A Complete Example

Putting together all the mechanisms described so far, we were able to develop an FPGA capable of self-replication and self-repair. In order to more clearly illustrate its features, in the next subsections we will provide a small but complete example, starting with a brief description of the function to be implemented in our FPGA and proceeding with a step-by-step analysis of the system's operation.

4.4.1 Description

While our final objective is to use MuxTree as a programmable platform to implement an array of artificial cells [92], describing such an implementation in detail would be extremely complex. On the other hand, it is possible to illustrate most or all of the novel features of our FPGA on a much simpler example.

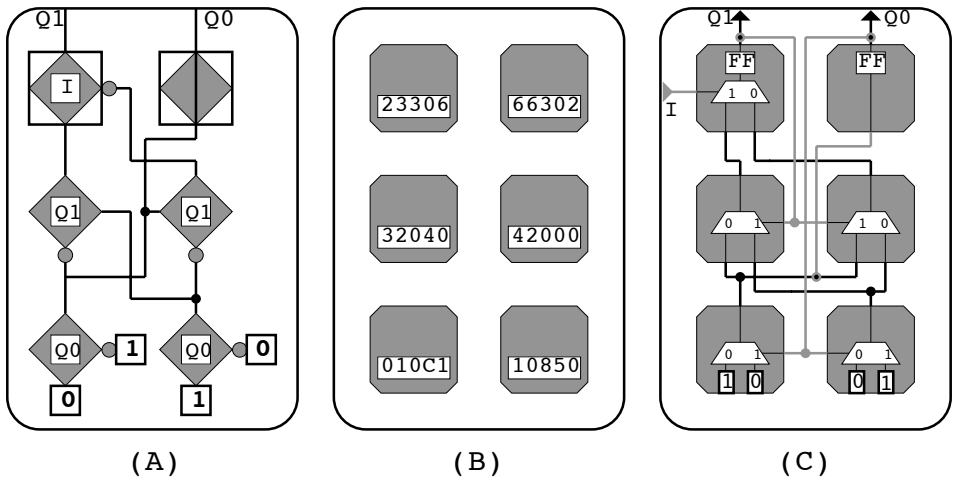

Figure 4-21: Binary decision diagram (A), configuration (B), and connection network (C) for the up-down modulo-4 counter.

To prove the validity of our approach, we developed a prototype of our FPGA (which we called *MuxTreeSR*) in the form of a set of 18 interconnecting modules, each containing a single MuxTree element (see Appendix B). For demonstration purposes, we designed a simple application which we successfully implemented on our prototype: an up-down modulo-4 counter.

From the function's BDD we easily derived the configuration of the six elements required by this application (Fig. 4-21). The configuration of the modulo-4 counter consists of 6 active elements, four of which are purely combinational (the bottom two rows), while two are sequential (the top row), that is, make use of the internal flip-flop to store the current state of the counter. In addition, a single external input `I` provides the control variable which determines the direction of the counter: if `I=0` the circuit will count up (i.e., repeat the sequence `0→1→2→3→0`...), while if `I=1` it will count down (i.e., `3→2→1→0→3`...).

4.4.2 The Self-Replication Phase

Arranging our 18 elements in a 6x3 array (Fig. 4-22) allows us to implement either three identical copies of the counter with no spare elements or, more usefully to demonstrate self-repair, two copies with one spare column each[12].

To realize two copies of our counter, we will therefore use our self-replication mechanism to partition the array into two blocks of 3x3 elements, the third column being spare. The first step in the configuration of the FPGA is to input the appropriate sequence of states to the element in the southwest corner of our cellular automaton. If we assign the value `0` to a junction state, `1` to a vertical wall state, `2` to a horizontal wall state, and `3` to a spare column state, the required sequence is shown in Fig. 4-22. After the automaton has finished propagating, the array has been partitioned as desired (Fig. 4-23).

12. Of course, we could also implement a single copy of the counter with two spare columns, which would maximize the robustness of the system, but would imply that two columns in our prototype would be "wasted".

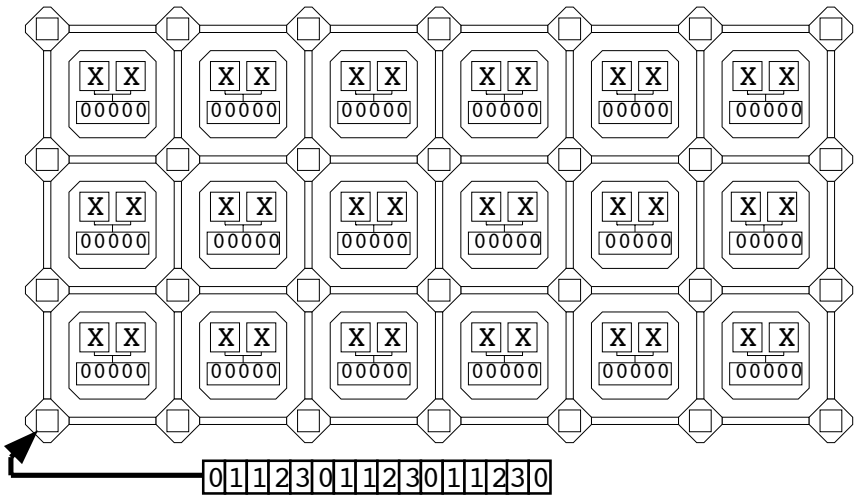

Figure 4-22: An empty 6x3 array of MuxTree elements and the sequence of states required to partition it for the modulo-4 counter.

4.4.3 The Configuration Phase

Before entering the actual configuration bitstream, the test sequence `180001` (i.e., the test pattern described above in subsection 4.2.5) is shifted into all the configuration registers in parallel to verify that no fault is present. As an example, let us assume that one of the registers (2nd row, 4th column) contains a stuck-at-0 fault (Fig. 4-24). The fault will prevent the leading `1` from reaching the end of the register, and the resulting `C0000` sequence generates a fault detection signal. Since the configuration has not yet been entered, there is no need at this stage for explicit reconfiguration: the element simply becomes `DEAD` (a change of internal state which requires a single clock period), which automatically causes a rerouting of the connections.

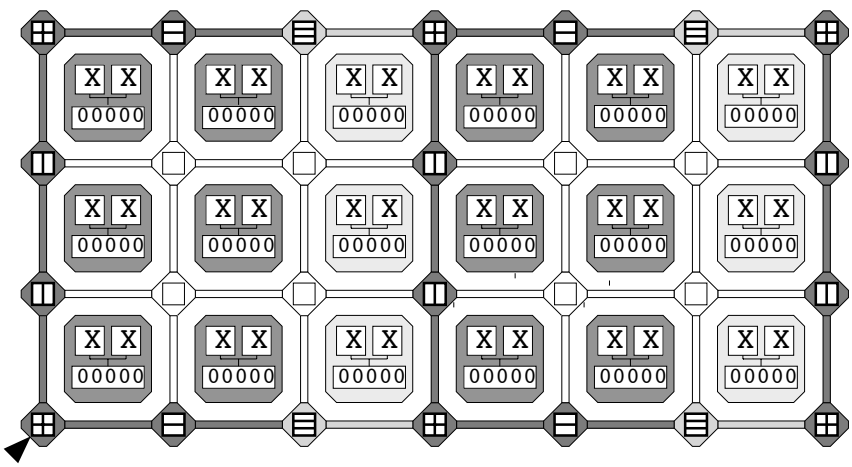

Figure 4-23: The array after partitioning.

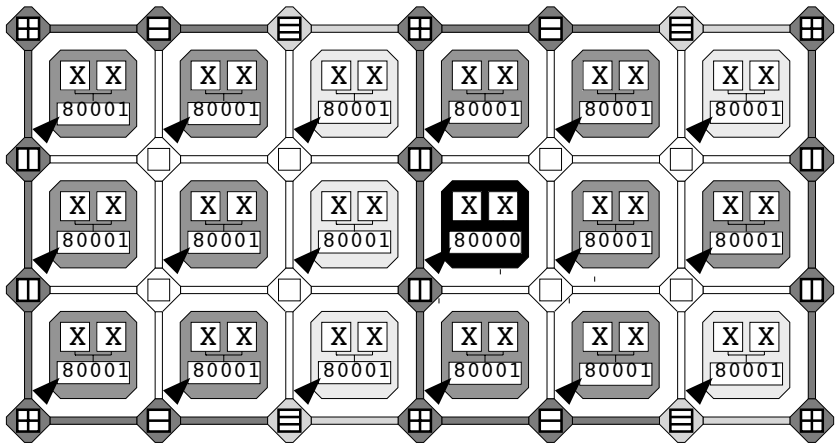

Figure 4-24: The test sequence detects a faulty register.

Once the registers have been tested, the actual configuration is shifted into each block in parallel starting from the element in the southwest corner (the entry point), and uses the membrane to determine its own propagation path (Fig. 4-25). When the bitstream encounters the dead element, it is automatically (and transparently) redirected to its eastern neighbor, whose configuration will in turn be redirected to the appropriate spare element.

What occurs is therefore an implicit reconfiguration of the array: rather than actively shifting the configuration of the faulty element, the bitstream is automatically rerouted so as to exploit the spare element.

Once the six active elements have been configured (there is no need to provide a configuration for the spare elements, and thus no need to modify the bitstream depending on the frequency of spare columns), the circuit is ready to operate.

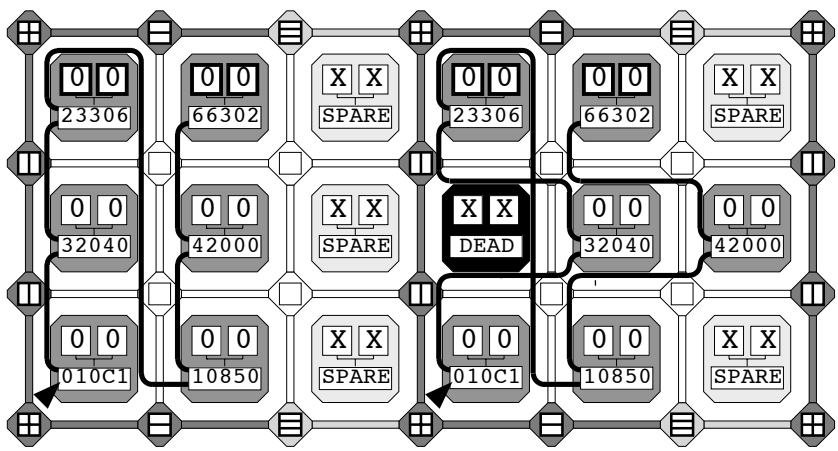

Figure 4-25: Propagation of the configuration in the presence of a faulty register.

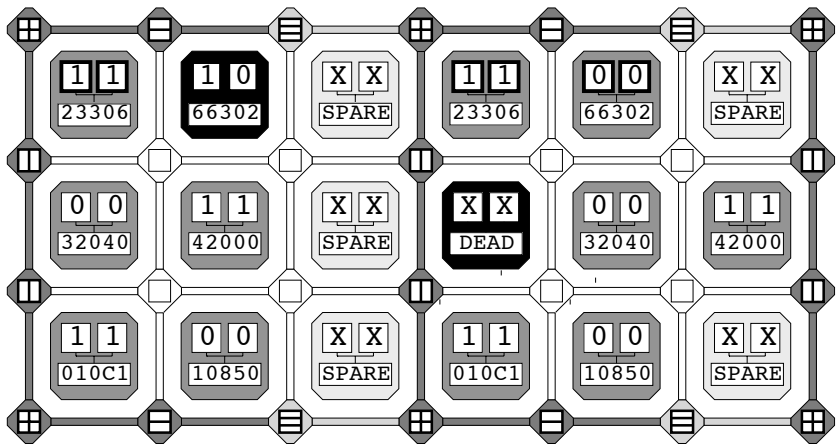

Figure 4-26: A fault is detected in one of the elements during operation.

4.4.4 The Operating Phase

Should a fault be detected while the circuit is operating, the reconfiguration mechanism will be activated. For example, let us assume that a stuck-at-1 fault is detected (by comparison) in the left copy of the functional part of the element located in the 3rd (top) row and 2nd column of the array (Fig. 4-26). As soon as the fault is detected, the circuit is frozen (i.e., the functional clock is disabled), and the self-repair mechanism starts shifting the configuration of the dying element to the spare element to its left. Once the shift is complete, the circuit is reactivated by once again enabling the functional clock (Fig. 4-27). Since, as we have seen, the contents of the flip-flop in the faulty cell are chained to the contents of the register in the shift, the circuit will resume operating from the state it was in when the fault was detected (in this case, the count was stopped at 2 by the reconfiguration).

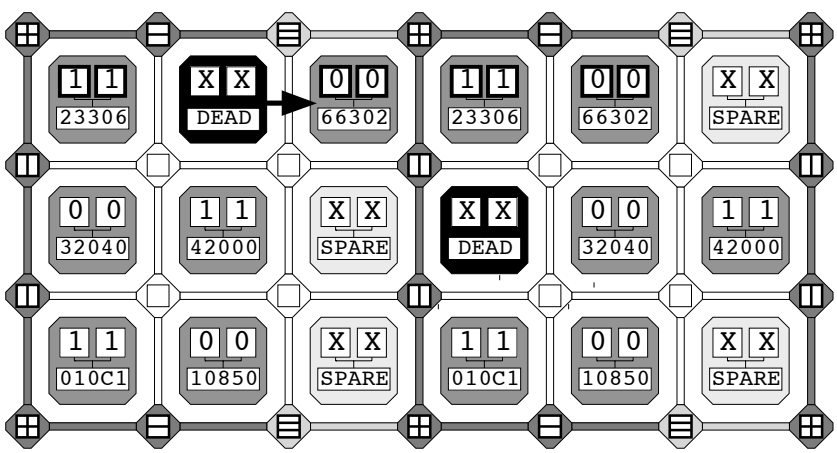

Figure 4-27: The faulty element is replaced by a spare and the array resumes operation.

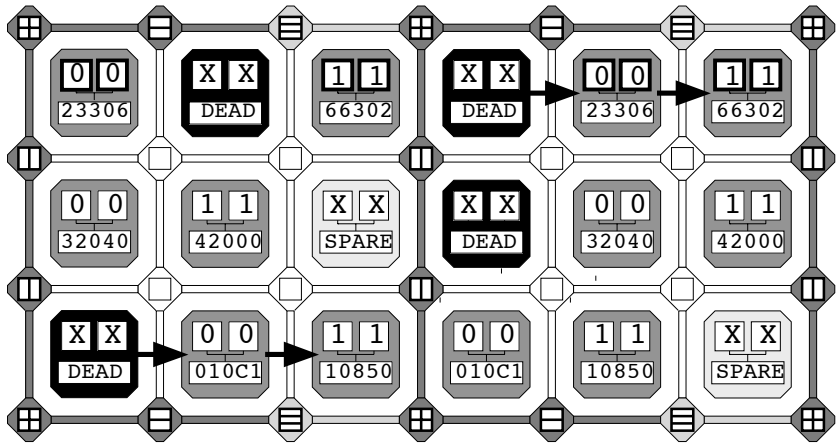

Figure 4-28: Reconfiguration of the array in the presence of multiple faults.

The available spare elements allow for further faults to be repaired (Fig. 4-28), but obviously the reconfiguration capabilities can be saturated (for example, if a fault were to be detected in the element in the 2nd row and 5th column). In this case, as we mentioned, we condemn an entire block by propagating a KILL signal and rely on a second level of redundancy, which can consist either of a relatively complex reconfiguration mechanism such as that exploited in our artificial cells or, as in this case, of the much simpler duplication of the counter implicit in the self-replication mechanism (Fig. 4-29).

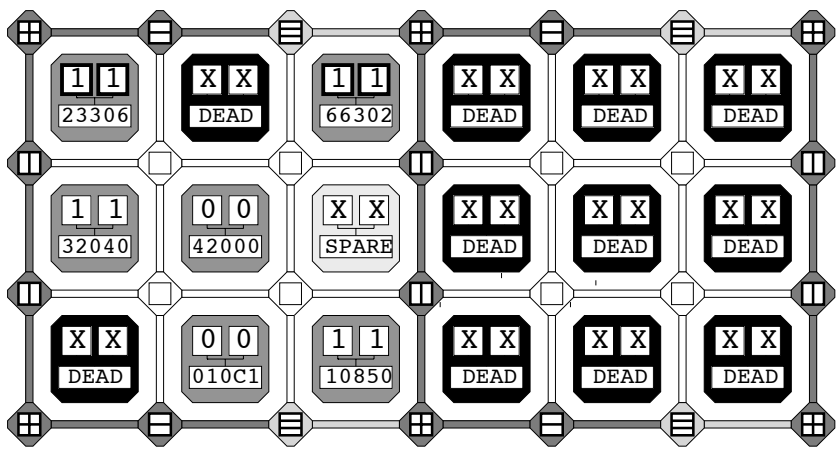

Figure 4-29: A block of elements, and thus one of the counters, is killed when the reconfiguration mechanism is saturated. The second copy continues to operate.

CHAPTER 5

CONCLUSION

In this conclusion, we will try to consider to what extent the system we have presented fulfills the initial goals of the project (section 5.1), as outlined in the introduction to this thesis, and specify the original contributions we had to bring to the state of the art in order to accomplish these tasks (section 5.2). We will then analyze our self-repairing FPGA in the light of a possible utilization *outside* the Embryonics project (section 5.3), and we will conclude by discussing possible future developments both for MuxTree in particular and for the Embryonics project in general (section 5.4).

5.1 Analysis of the Results

Our goal in this thesis was to design an FPGA capable of self-replication and self-repair. The particular requirements of such features were introduced in section 1.2. In this section, we want to compare our results with the original requirements.

As far as self-replication is concerned, our goal was met in most respects. Our self-replication mechanism is indeed capable of generating multiple copies of our artificial cells from the description of a single such cell. The mechanism allows for cells of any given size, and thus capable of executing any given task. The only compromise we had to accept was the use of an external source for the configuration of the cells. As we mentioned in the introduction, ideally it should be the cells themselves which generate and control the replication process. In our case, the replication process is indeed controlled by the dedicated hardware integrated in our FPGA, but the configuration bitstream is generated outside the circuit itself and not by the cells. The development of such an "ideal" system remains a future research goal for the Embryonics project.

As far as the self-repair mechanism is concerned, the results are not quite as close to optimum. Notably, the self-test mechanism falls somewhat short of the ideal outlined in the introduction. The constraints of biological inspiration, coupled with the need to minimize the hardware overhead, proved too strong to allow on-line self-test of more than a relatively small part of the circuit. However, our system is indeed able to transparently detect faults on a large part of the circuit[1] through off-line self-test at configuration.

On the other hand, the self-repair mechanism itself fits our requirements remarkably well: it allows for the repair of a considerable number of faults (the exact coverage depends, of course, on the number of spare elements allotted to the array) and is indeed capable of activating the self-repair at the cellular level through its global KILL signal. While the self-repair process is not guaranteed to occur transparently to the user, an effort was made to minimize the time it requires. In addition, our system actually surpasses our requirements by introducing the very useful feature of programmable redundancy, which allows the user to determine the amount of logic to be "sacrificed" for additional self-repair capabilities.

As a final consideration, we will mention that the hardware overhead required by the introduction of self-repair and self-replication is of approximately 50-70%[2] compared to the basic version of MuxTree. Considering the extremely fine grain of our FPGA, we are very satisfied by this figure, which includes all the additional logic necessary for self-replication, self-test, and self-repair, as well as the control logic required to handle these processes.

5.2 Original Contributions

When faced with a thesis such as this one, describing a research effort which is closely integrated within a larger project, it is sometimes difficult to precisely identify the original contributions of the author. In this section I will try to point out my personal contributions to the project within each of the main chapters.

Chapter 2 is meant to provide some background material for the Embryonics project in general. As such, I obviously cannot claim sole credit for its development: in particular, the epigenetic and phylogenetic axes are not really within the scope of my research. As far as the ontogenetic axis is concerned, its development was a collective effort on the part of a small group of people, including myself. I feel I have contributed in a substantial way to its development, and in particular to the definition of the 3-layer system (organism, cell, molecule) which is the core of our vision of ontogenetic hardware. Obviously, my contribution was mostly centered on the definition of the requirements and constraints of the molecular layer.

Chapter 3 contains what is probably my most original contribution, at least from a conceptual standpoint: the self-replication mechanism. When I first approached the problem, the state of the art for self-replicating machines was represented by Langton's loop, a structure obviously ill-suited to a hardware realization. In a first phase, I therefore had to improve the state of the art by design-

1. We cannot provide an accurate estimate of the fault coverage provided by our system, as MuxTree is a circuit in constant evolution still far from its final implementation.

2. Again, an accurate estimate is difficult, both because MuxTree is constantly evolving and because our only physical realization (described in Appendix B) relies on programmable logic (Xilinx FPGAs) to implement our elements. A more accurate estimate would be possible if MuxTree were to be implemented as a VLSI chip.

ing a novel self-replicating loop which, while not directly designed for hardware implementation, represented an important step forward in the development of computationally-useful self-replicating structures[3]. The need to design new self-replicating structures also led to the development of a novel software tool (see Appendix A) which provides a novel approach to the design of complex cellular automata. In a second phase, I had to develop an hardware mechanism capable of implementing self-replication in our FPGA. This process, which led to the integration of the cellular automaton in the MuxTree array, required considerable original thought since, to the best of my knowledge, such a mechanism is quite unique.

Chapter 4 deals with the hardware implementation of the self-test and self-repair mechanisms on the MuxTree FPGA. The actual implementation, in the form of a digital logic circuit programmed into a Xilinx FPGA (see Appendix B), was entirely my own work. As far as the architecture of the system is concerned, determining the original features of the design is less straightforward. The programmable function and the switch block of the MuxTree element predate my arrival in the laboratory, and while I was forced to introduce some minor modifications, I cannot claim authorship. On the other hand, the configuration mechanism is entirely my own work. For self-test and self-repair, I was forced to rely on standard techniques, mostly because the size of the elements did not allow complex mechanisms. However, while the test and repair strategies are not original (comparison, test patterns, spare columns, etc., are all "standard" techniques), it was, to the best of my knowledge, the first attempt to integrate on-line self-repair in a circuit as fine-grained as MuxTree. An important effort was thus required in order to select which approaches were viable given our constraints. I also had to integrate the mechanisms into the existing hardware with a major effort towards minimizing the additional silicon. This effort was remarkably successful in the case of the self-repair mechanism (which exploits much of the logic already in place). A notable original achievement was also the idea of exploiting the cellular automaton to configure the degree of fault tolerance (to the best of my knowledge, this feature is also unique, at least where FPGAs are concerned).

5.3 MuxTreeSR outside of Embryonics

When considering possible applications for our FPGA outside of the Embryonics project, we must remember that MuxTreeSR was conceived as part of a larger academic research project. As such, several factors which would be crucial in the development of a commercial circuit (such as, for example, hardware overhead or speed of operation) were, if not ignored, at least given a lower priority

3. The gratifying number of references to this work in the literature seems to indicate that the automaton, designed as an intermediate step, has nevertheless a certain intrinsic interest.

with respect to other constraints related to the overall inspiration of the Embryonics project.

The net result of this approach is that MuxTreeSR, as a whole, would probably not be a commercially viable product. However, this consideration does not preclude the possibility of adapting the particular *mechanisms* we developed in this project to commercial systems. In this section, we will analyze the strength and weaknesses from a commercial standpoint of three separate parts of our system: the MuxTree FPGA, the self-replication mechanism, and the self-repair approach.

5.3.1 MuxTree

The most recent developments in the design of FPGAs seem to indicate a transition from general-purpose programmable logic arrays towards circuits which are adapted to certain specific applications [66, 72]: digital signal processing (DSP), control tasks, mathematical coprocessing, etc. This is somewhat of an advantage when considering possible commercial applications for MuxTree, an FPGA which is not well suited for certain tasks. In particular, its extremely fine grain and its homogeneity (which prevents the use of wide long-distance busses) is a weakness for applications which handle large (32 or 64 bits) data. Also, MuxTree is at a disadvantage in implementing complex mathematical operators compared to many existing commercial FPGAs which integrate support for such operators in the structure itself of their elements.

On the other hand, the structure of MuxTree, designed to efficiently implement binary decision trees and diagrams, could be an interesting advantage for applications which can be easily described as logic functions (e.g., many state machines and control applications). A considerable number of software packages, both commercial and academic, are capable of deriving minimized binary decision diagrams from a given logic function, and such diagrams can then be used to trivially generate the configuration for a MuxTree array. Since many such control-applications are not usually speed-critical, MuxTree's shortcomings in this respect become less important.

These considerations, however interesting and, in a sense, useful, should not be given too much emphasis: MuxTree was not designed to be and is not likely to become a commercial product in the foreseeable future. Even if it has the potential to become an useful programmable logic device outside of the Embryonics project, it does not, by itself, provide enough unique advantages to be able to compete with the latest generation of FPGAs. Moreover, adapting it for a commercial release would require a major effort on the part of a team of developers, a task outside the competence of our laboratory. In particular, the development of the software tools which would be indispensable to achieve commercial success, and which are basically non-existing at this stage, is an effort beyond our possibilities (and, to a large extent, outside of our interests).

5.3.2 Self-Replication

If MuxTree is not likely to be of interest to FPGA manufacturers in the foreseeable future, we feel this might not be the case for our self-replication mechanism.

At first sight, self-replication is the feature of our system which is most closely related to the particular requirements of the Embryonics project, and consequently the least likely to be of use outside of our project. However, if we look at our mechanism without considering the biologically-inspired cellular level, we can see that self-replication is an extremely efficient approach to the implementation of arrays of identical elements of any given structure.

We have mentioned that FPGAs appear to become more and more application-specific, and it is therefore not difficult to imagine FPGAs designed specifically to implement arrays of identical processing elements, structures which are very well-suited for a wide number of applications: SIMD (Single-Instruction Multiple-Data) parallel processing, bit-slice architectures, etc. Considering the complexity of such applications, and consequently the difficulty of configuring FPGAs to implement them, the possibility of automatically obtaining two-dimensional arrays of processing elements from the configuration bitstream of a single such element could become a very powerful advantage.

5.3.3 Self-Repair

We have already mentioned the interest of self-repair in the development of complex FPGAs [7, 8, 25, 35, 40, 51, 85] in the previous chapter. Applying such a mechanism to a commercial system is, of course, a complex task.

From a manufacturer's point of view, our system is probably *too* powerful: on-line self-test and self-repair are not yet enough of a priority to warrant the hardware overhead required by such systems. On the other hand, there is no reason why our mechanism might not be simplified in order to more closely fit the requirements of the commercial world. In particular, we feel that a simplified version of our system might very well be adapted to achieve self-repair at fabrication (a more likely requirement for FPGA manufacturers) with a more than acceptable overhead.

In the design of such a system, several modifications would be needed. In the first place, the self-test mechanism would have to be completely redesigned, both to be adapted to the new architecture of the elements and to take advantage of the possibility of operating off-line. While the test of the configuration register would easily be adapted to most new architectures, the test of the functional part and of the connections would have to be modified depending on the layout of the element.

As for the self-repair mechanism, very little modification would be required, as it already supports static reconfiguration (rerouting of the connections before configuration in case of permanent faults). In order to adapt it to a new architec-

ture for repair at fabrication, the only major alteration would be a simplification: the removal of the logic required for dynamic reconfiguration (rerouting of the connections and shift of the configuration whenever a fault is detected during operation).

Of course, the programmable redundancy of our system depends on the homogeneity of the array and on the self-replication mechanism, and would therefore be lost in a commercial system which is not likely to implement these two features. Nevertheless, the simplicity of the system, coupled with its versatility, could be of interest to FPGA manufacturers, even in a simplified form.

5.4 Embryonics: the Future

This thesis represents a step forward in the realization of the Embryonics project. However, it does not by any means represent a closure for the project as a whole, or even for the development of the ontogenetic axis. Research is continuing along all three axes of the POE model.

In the phylogenetic axis, where the design of Firefly [33] demonstrated the feasibility of hardware evolution, we are currently studying the application of evolutionary strategies to the design of hardware systems such as, for example, fuzzy controllers. As for future developments of evolutive hardware systems, we are investigating the feasibility of *open-ended undirected evolutionary strategies*, that is, systems which evolve not towards a precise, user-defined goal, but independently. Such an approach is undoubtedly a much closer approximation of natural evolution.

The epigenetic axis is advancing into the application phase, where we are trying to apply our algorithms to the solution of real-life problems and for the control of autonomous robots. An interesting possible evolution along this axis would be the creation of neural networks capable of *continuous learning*, that is of learning new behaviors (and consequently of adapting their structure) not only during a dedicated learning phase, but also while operating. Obviously, such systems would much more closely approach the behavior of biological neural networks than conventional ANNs.

On the ontogenetic axis, to which this thesis belongs, even if a considerable amount of work remains, we can begin to glimpse a possible closure. The next major step in the development of this axis is the design of the BioWatch 2001, an extremely complex machine which we hope to present on the occasion of the Expo.01, a major scientific and cultural event which will take place in the year 2001 in Switzerland. The function of the machine will be that of a self-replicating and self-repairing watch, implemented through macroscopic versions of our artificial cells and molecules (Fig. 5-1).

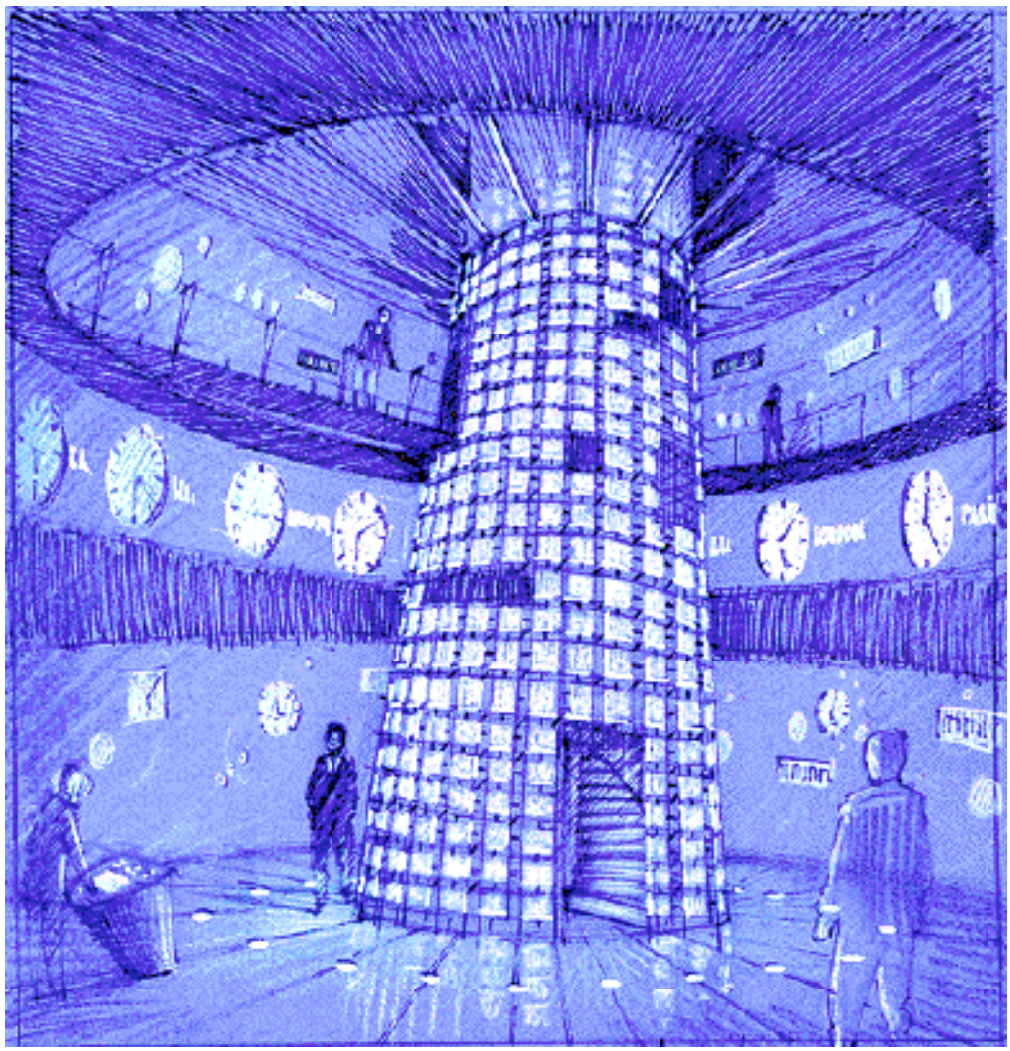

Figure 5-1: An artist's rendition of a possible realization of the BioWatch 2001.
[Art by Anne Renaud]

In addition to all the mechanisms described in this thesis, this machine, which will occupy an entire building, will require further improvements to the molecular level. Notably, in order to be efficiently used to create the kind of artificial cells required by the BioWatch 2001, MuxTree will require a mechanism allowing the configuration register to be used as a memory, accessible to the rest of the circuit. In fact, the direct consequence of our cellular approach is that the genome memory (the memory containing the program to be executed in each processor) is necessarily large, as it must contain the instructions to be executed in all processors in the array. In the current version of MuxTree, which provides a single flip-flop per element as memory storage, such a memory would require an excessive number of elements. By using the 20-bit configuration register for memory storage, the size of our artificial cells can be considerably reduced.

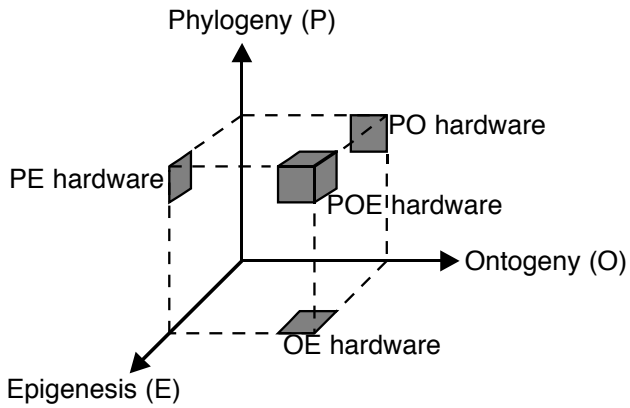

Figure 5-2: Convergence of the POE model.

Once this improvement is in place, the development cycle for MuxTree will approach its end. We might consider adding a few extra features, for example, in order to achieve true self-replication (as mentioned above in section 5.1), but such additions are not likely to improve the circuit's performance in any significant way. At this stage, we might well be interested in designing a VLSI chip containing an array of MuxTree elements.

The closure of development of MuxTree will not, however, necessarily mean the end of research in the ontogenetic axis. In fact, with a set of mechanisms in place capable of realizing an ontogenetic machine, we could try to consider a possible merging of the three axes of the POE model (Fig. 5-2). For example, following von Neumann's sequence of self-replicating machines, we could imagine replacing the functional part of the MuxTree element with a neuron-like structure, thus joining the epigenetic and ontogenetic axes.

For the moment, this kind of convergence is fairly remote, and a subject of speculation only. On the other hand, the work presented in this thesis is an interesting first step in the development of such advanced systems. By introducing features such as self-replication and self-repair, we hope to have shown that it is possible to draw inspiration from biology in the design of digital circuits, and indeed that bio-inspiration can lead to the development of novel and powerful architectures.

APPENDIX A

CAEDITOR

The design of our novel self-replicating automaton (subsection 3.4.3) posed a considerable technical challenge, mostly because hardly any existing software package is well suited to the task. Most "conventional" applications of cellular automata (e.g., the modelization of physical or chemical processes) assume that the transition rules governing the behavior of the automaton are known, or in any case can be determined independently of the automaton itself. The design of a self-replicating automaton, on the other hand, requires that the rules be determined so as to model a given process (self-replication).

In order to efficiently investigate different self-replication strategies, we needed a tool which would allow us to easily alter or redesign complex structures (e.g., loops) capable of complex behavior. After experimenting with existing design methods, we decided to develop our own software package, known as *CAEditor*. In this appendix we will describe the operation of CAEditor (section A.1), introduce the transition rules of some of the automata we discussed in the previous chapters (section A.2), and conclude with a few technical considerations and the instruction on how to obtain a copy of our software (section A.3).

A.1 The CAEditor Design Tool

In this section, we will introduce the main features of our software tool, starting with a general overview of the system (subsection A.1.1), through a basic tutorial on its use (subsection A.1.2), and finishing with a brief description of some more "advanced" features (subsections A.1.3 and A.1.4).

A.1.1 Overview

The main screen of CAEditor (Fig. A-1) can be subdivided into four areas: the *configuration area*, the *cellular space area*, the *state area*, and the *rule area*.

The configuration area displays the basic parameters of the automaton: the number of states, the size of the cellular space, the size of neighborhood, the initial configuration, etc. (for more details, see the sample configuration file in section A.2). It also allows the user to modify these parameters and save/load a configuration to/from a file.

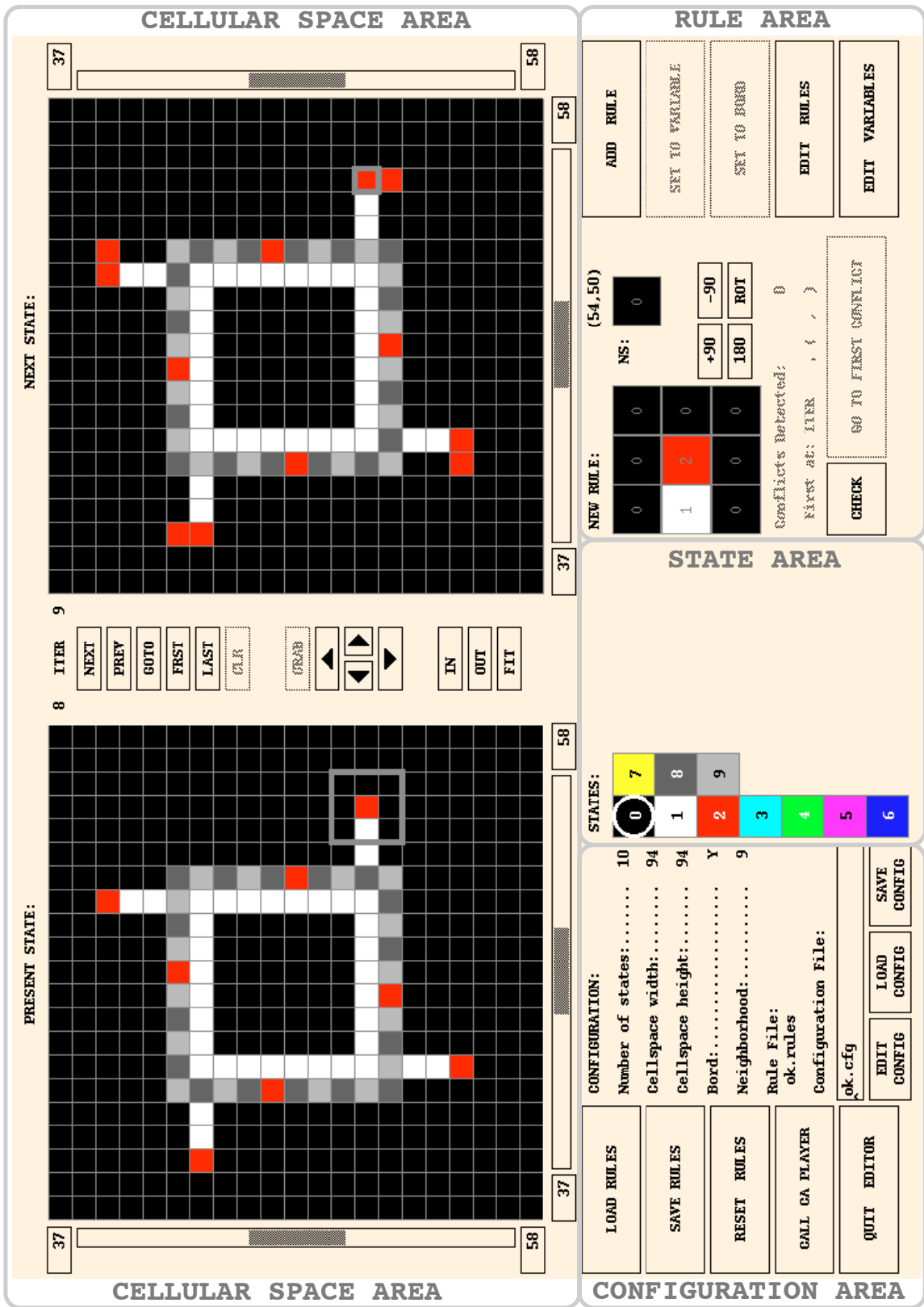

Figure A-1: CAEditor's main screen.

This area also contains a set of buttons allowing the user to load/save a set of transition rules from/to an external file (see the sample rule files in section A.2), as well as to reset the rules currently in use. Also, a button allows the user to quit the CAEditor program.

The `CALL CA PLAYER` button launches a subprogram (the *CA player*) dedicated to the high-speed execution of the automata developed using the editor. Unlike the main program, which is optimized for design, the player is heavily optimized for execution: it accesses the same data structures as the main program, but exploits a high-performance engine to determine and display the next state of the elements.

The cellular space area is the heart of the system, and reflects the philosophy underlying our approach to the design of complex automata. Since the aim of our tool is to allow the user to find the rules allowing an automaton to behave according to a predetermined process (e.g., self-replication), we developed an approach based on an incremental definition of the transition rules. In order to clarify this concept, let us examine the contents of this area.

On the left, a display shows the present state of the automaton, that is, the state of its elements at a given iteration N. This display is passive: it cannot be modified. On the right, a second display shows the next state of the automaton, that is, the state of its elements at iteration N+1, *given the current transition rules*. Unlike the present state, the next state can be modified by the user. By imposing a given state (selected in the state area) to any of the elements in this display, the user can automatically add the corresponding rule to the transition table (for a more detailed description of this process, see the next subsection). The user can thus define the rules which implement a given process through incremental steps (iteration by iteration).

Between the two main displays, a set of buttons allow the user to advance to the next iteration, come back to the previous one, etc., as well as perform the standard graphical operations on the display (zoom in, zoom out, move in the four directions, etc.).

The state area displays the available states, and is used to select an "active" state which will be applied in order to find a given rule (as seen above).

Finally, the rule area is used to handle all the operations related to the insertion of a rule in the transition table. On the top left, a display shows the present state of the chosen element's neighborhood (9, as in this case, or 5), alongside the selected next state. Both displays are active, and can be manually altered by the user, who can set any one of the elements either to an available state, including the special state which represents the border of the cellular space (`SET TO BORD`), or else to a state variable (`SET TO VARIABLE`, see subsection A.1.4 below). A set of four buttons allows the user to automatically or manually add the rotated versions of the transition rule.

Selecting a next state for a given neighborhood is, unfortunately, not the only operation required to add a rule to the transition table. In fact, before a rule can be added (`ADD RULE`), it is necessary to check that no *conflict* exists with previously-inserted rules, that is, that the addition will not invalidate one of the rules which have already been used in the development of the automaton. This is the function of the `CHECK` button. Whenever the user attempts to add a rule, the program will examine all the iterations leading to the present state to ensure that the selected neighborhood has never occurred in the past. If it finds that the rule is "safe" (that is, that it will not generate any conflict in past iterations), the rule is added to the transition table, and the displays in the cellular space area are updated. If, on the other hand, the program finds that the selected neighborhood has already occurred in earlier iterations, it will display the number of the first such iteration, as well as the coordinates of the conflict area. At this point, the user can either give up the attempt to add the new rule, or else access the conflicting iteration (`GO TO FIRST CONFLICT`).

Finally, the last two buttons in the rule area allow the user to access two subprograms: the *rule editor* (subsection A.1.3) and the *variable editor*, which allow the definition of state variables (subsection A.1.4).

A.1.2 Operation

The design of a new automaton begins with the definition of the global parameters (number of states, neighborhood, etc.). These parameters can be altered at a later stage, so it is not necessary to know the correct values at the beginning of the design: generally, it is better to begin with the lowest possible values for the number of states and especially the size of the array, which have a major impact on the amount of memory required by the program and on its speed of execution.

The next step is to define the initial configuration, that is, the state of the elements at iteration 0. This operation is performed by selecting a state and using the cursor in the cellular space area to assign it to the appropriate elements in the cellular space (it might be useful, depending on the size of the space, to zoom in on a particular area).

At this point, all the data required by the program to begin operating are present, and it is advisable to save the configuration to file.

Advancing to the next iteration, the cellular space area will display two identical copies of the initial configuration: since no transition rules have been specified, the program assumes that all elements will remain in the same state.

The user can now begin to modify the next state display to obtain the desired behavior: selecting a state and clicking on an element in the display (again, the zoom might be useful) will cause a new transition rule to appear in the rule area. An attempt to add the rule will then undoubtedly succeed, since no conflict can be generated (no other rule is present).

As soon as the rule is added to the transition table, the cellular space area is updated to reflect the change, displaying the updated present and next states. The user can then keep inserting additional rules, and the program will automatically assure that no conflict exists.

Once the first iteration has been completely defined, the user can advance to the next, add the rules necessary to implement the desired behavior, and so on. The user can thus iteratively define the states of the automaton until the transition table has been completely defined.

A.1.3 The Rule Editor

In certain cases, and notably if a set of rules is known to be correct, it might be desirable to insert the rules directly, without going through the graphical interface and the verification process.

To this end, we introduced a sub-program, known as the *rule editor*, which allows the list of transition rules to be accessed directly: the user can than add, remove or find rules directly from the list without going through the "standard" process.

The rule editor is a powerful but "dangerous" tool, as it allows rules to be added and discarded without considering the effect of such operations on the automaton: there is no guarantee that no conflicts will be generated or that all necessary rules will be preserved.

Also accessible through the rule editor is the *cleanup* feature. When this process is activated, the program will analyze all the current transition rules and discard all those which are either *covered* (i.e., superseded by other rules) or redundant. In the process of designing of complex automata, such rules are anything but rare, often a consequence of run-time modifications to the behavior of the automaton.

A.1.4 State Variables

A peculiar feature of our tool is the use of *state variables*, that is, symbols (usually letters of the alphabet) used to represent a set of states. For example, suppose we define two variables

```
A={1,2}
```

and

```
B={1,3,4}
```

and assume that rules for a neighborhood of 5 are defined as

```
PS,N,E,S,W=NS
```

where `PS` is the present state of the element, `N`, `E`, `S`, and `W` are the present states of the neighbors to the north, east, south, and west respectively, and `NS` is the next state of the element.

Then the rule

```
A,0,B,A,0=1
```

will expand into the following rules:

```
1,0,1,1,0=1
1,0,3,1,0=1
1,0,4,1,0=1
2,0,1,2,0=1
2,0,3,2,0=1
2,0,4,2,0=1
```

Variable are a useful tool for achieving a more compact representation of the transition rules of an automaton (for example, they are very helpful in designing loops and dynamic data storage in general), but should not be abused: a transition rule containing 4 different variables of 5 states each will expand to 5^4=625 actual rules!

A.2 Sample Transition Tables

CAEditor requires two files for each automaton. The first is a *configuration file*, containing general information such as the number of states, the size of the cellular space, the size of the neighborhood (5 or 9), as well as a list of all the elements which are not in the quiescent state at iteration 0 (i.e., the *initial configuration* of the automaton).

As an example, Table A-1 shows the configuration file for our self-replicating loop (see Fig. 3-13). It defines the number of states (`NST`), the size of the cellular space (`CSX` and `CSY`), and the neighborhood (`HOOD`). It also defines (`BORD`) whether the cellular space should be considered bounded (i.e., have borders) or toroidal.

NST=7;	(42,49)=5;	(44,43)=1;	(48,42)=6;	(50,49)=1;
CSX=94;	(42,50)=6;	(44,50)=1;	(48,43)=1;	(50,50)=1;
CSY=94;	(42,51)=5;	(44,51)=5;	(48,50)=1;	(50,51)=5;
BORD=Y;	(43,41)=1;	(45,42)=5;	(48,51)=5;	(50,52)=1;
HOOD=9;	(43,42)=5;	(45,43)=1;	(49,42)=5;	(51,42)=5;
RULEFILE=”ok.rules”;	(43,43)=1;	(45,50)=1;	(49,43)=1;	(51,43)=6;
REPORTFILE=”ok.report”;	(43,44)=1;	(45,51)=6;	(49,50)=1;	(51,44)=5;
(41,50)=1;	(43,45)=1;	(46,42)=6;	(49,51)=6;	(51,45)=6;
(42,42)=6;	(43,46)=1;	(46,43)=1;	(50,42)=6;	(51,46)=5;
(42,43)=5;	(43,47)=1;	(46,50)=1;	(50,43)=1;	(51,47)=2;
(42,44)=6;	(43,48)=1;	(46,51)=2;	(50,44)=1;	(51,48)=5;
(42,45)=5;	(43,49)=1;	(47,42)=2;	(50,45)=1;	(51,49)=6;
(42,46)=2;	(43,50)=1;	(47,43)=1;	(50,46)=1;	(51,50)=5;
(42,47)=5;	(43,51)=6;	(47,50)=1;	(50,47)=1;	(51,51)=6;
(42,48)=6;	(44,42)=6;	(47,51)=6;	(50,48)=1;	(52,43)=1;

Table A-1: The configuration file for our self-replicating loop.

The initial configuration is specified by a set of directives in the form

```
(X,Y)=S;
```

where `X` is the horizontal coordinate of the element in the cellular space, `Y` is its vertical coordinate, and `S` is its initial state.

The configuration file also contains the name of the *rule file*, a file containing the transition rules which govern the behavior of the automaton (the two files are kept separate to allow multiple initial configurations for a single set of transition rules).

As an example of a typical rule file, Table A-2 shows the transition rules for Langton's loop[1].

0,0,0,0,1=2;	0,1,2,5,2=5;	1,0,1,2,4=4;	1,1,2,2,7=7;	2,0,2,0,7=3;	4,0,1,2,5=0;	5,1,2,4,2=2;
0,0,0,0,6=3;	0,1,2,6,2=1;	1,0,1,2,7=7;	1,1,2,4,2=4;	2,0,2,3,2=1;	4,0,2,1,2=0;	5,1,2,7,2=2;
0,0,0,0,7=1;	0,1,2,7,2=1;	1,0,2,0,2=6;	1,1,2,7,2=7;	2,0,2,5,2=0;	4,0,2,2,2=1;	6,0,0,0,1=1;
0,0,0,1,1=2;	0,1,2,7,5=1;	1,0,2,2,4=4;	1,2,2,2,4=4;	2,0,3,2,1=6;	4,0,2,3,2=6;	6,0,0,0,2=1;
0,0,0,1,2=2;	0,1,4,2,2=1;	1,0,2,2,6=3;	1,2,2,2,7=7;	2,0,3,2,2=6;	4,0,2,5,2=0;	6,0,2,1,2=0;
0,0,0,1,3=2;	0,1,4,3,2=1;	1,0,2,2,7=7;	1,2,2,4,3=4;	2,0,5,5,2=1;	4,0,3,2,2=1;	6,1,2,1,2=5;
0,0,0,2,1=2;	0,1,4,4,2=1;	1,0,2,3,2=7;	1,2,2,5,4=7;	2,0,5,7,2=5;	5,0,0,0,2=2;	6,1,2,1,3=1;
0,0,0,2,6=2;	0,1,4,7,2=1;	1,0,2,4,2=4;	1,2,3,2,4=4;	2,1,1,2,6=1;	5,0,0,2,3=2;	6,1,2,2,2=5;
0,0,0,2,7=2;	0,1,6,2,5=1;	1,0,2,6,2=6;	1,2,3,2,7=7;	3,0,0,0,2=2;	5,0,0,2,7=2;	7,0,1,1,2=0;
0,0,0,5,2=5;	0,1,7,2,2=1;	1,0,2,6,4=4;	1,2,4,2,5=5;	3,0,0,0,4=1;	5,0,0,5,2=0;	7,0,1,2,2=0;
0,0,0,6,2=2;	0,1,7,2,5=5;	1,0,2,6,7=7;	1,2,4,2,6=7;	3,0,0,0,7=6;	5,0,2,0,2=2;	7,0,1,2,5=0;
0,0,0,7,2=2;	0,1,7,5,2=1;	1,0,2,7,1=0;	1,2,5,2,7=5;	3,0,0,4,2=1;	5,0,2,1,2=2;	7,0,2,1,2=0;
0,0,1,0,2=2;	0,1,7,6,2=1;	1,0,2,7,2=7;	2,0,0,0,7=1;	3,0,0,6,2=2;	5,0,2,1,5=2;	7,0,2,2,2=1;
0,0,2,1,2=5;	0,1,7,7,2=1;	1,0,5,4,2=7;	2,0,0,2,5=0;	3,0,1,0,2=1;	5,0,2,2,2=0;	7,0,2,2,5=1;
0,0,2,3,2=2;	0,2,5,2,7=1;	1,1,1,2,4=4;	2,0,0,3,2=6;	3,0,1,2,2=0;	5,0,2,2,4=4;	7,0,2,3,2=1;
0,0,5,2,2=2;	1,0,0,0,7=7;	1,1,1,2,7=7;	2,0,0,4,2=3;	3,0,2,5,1=1;	5,0,2,7,2=2;	7,0,2,5,2=5;
0,1,2,3,2=1;	1,0,0,2,4=4;	1,1,1,5,2=2;	2,0,0,5,1=7;	4,0,1,1,2=0;	5,1,2,1,2=2;	7,0,2,7,2=0;
0,1,2,4,2=1;	1,0,0,2,7=7;	1,1,2,2,4=4;	2,0,0,5,7=5;	4,0,1,2,2=0;	5,1,2,2,2=0;	

Table A-2: The 125 transition rules for Langton's Loop. [PS,N,E,S,W=NS;]

The transition rules for a neighborhood of 5 are in the form

```
PS,N,E,S,W=NS;
```

where `PS` is the present state of the element, `N`, `E`, `S`, and `W` are the present states of the neighbors to the north, east, south, and west respectively, and `NS` is the next state of the element.

Similarly, the transition rules for a neighborhood of 9 are in the form

```
PS,N,NE,E,SE,S,SW,W,NW=NS;
```

where `PS` is the present state of the element, `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, and `NW` are the present states of the neighbors to the north, northeast, east, southeast, south, southwest, west, and northwest respectively, and `NS` is the next state.

Finally, the configuration file specifies a *report file*, used by the program to print out information concerning its operation, such as a report of its activities, as well as eventual error messages.

1. Note that, for this automaton as well as for all others, rotated rules have been removed from the table.

VAR a = {5,6};	0,2,0,0,0,0,2,1,1=3;	3,3,0,0,0,0,0,1,1=0;	a,0,b,1,1,c,3,0,0=b;	b,0,0,0,c,1,a,0,1=c;
VAR b = {5,6};	0,0,0,0,0,3,3,0,0=3;	3,0,1,0,0,3,1,0,1=1;	b,0,3,1,1,1,a,0,0=3;	b,1,c,d,1,1,a,0,0=c;
VAR c = {5,6};	3,0,0,0,0,0,0,1,3=0;	0,1,3,0,0,0,0,1,1=3;	2,c,1,1,a,0,1,3,b=b;	1,3,1,0,0,0,4,a,b=4;
VAR d = {5,6};	3,0,0,0,3,0,1,0,0=1;	3,1,0,0,0,0,0,1,1=0;	a,0,b,1,2,1,3,0,0=b;	b,0,c,1,1,4,a,0,0=c;
2,1,b,0,0,0,a,1,1=a;	0,0,0,0,0,0,0,1,3=3;	3,1,0,0,0,0,2,1,1=2;	3,0,b,2,1,1,a,0,0=b;	b,0,c,b,1,4,a,0,0=1;
a,0,0,0,b,1,1,1,2=b;	0,3,3,0,0,0,2,1,1=2;	2,1,3,0,0,0,0,1,1=3;	3,0,b,1,1,1,a,0,0=b;	1,c,b,a,0,0,4,a,b=0;
b,0,0,1,c,1,1,1,a=c;	0,3,0,0,0,0,2,1,1=2;	3,1,2,0,0,0,0,1,1=0;	a,0,b,a,1,1,3,0,0=b;	a,0,2,1,b,0,1,0,0=2;
b,0,1,0,c,d,1,1,a=c;	3,0,3,2,1,0,0,0,3=2;	2,1,0,0,0,0,3,1,1=0;	2,0,a,1,1,0,0,0,0=a;	b,2,1,1,c,0,0,1,a=a;
b,1,0,0,0,0,c,1,a=c;	3,0,0,3,2,0,0,0,0=1;	0,0,a,b,1,1,3,0,0=3;	a,0,b,1,1,0,2,0,0=b;	1,2,a,b,0,0,0,0,0=2;
c,a,b,0,0,0,d,1,1=d;	0,3,2,1,0,0,0,0,0=3;	a,0,2,1,b,1,3,0,0=2;	2,0,a,1,0,1,0,0,0=a;	2,a,2,b,0,0,0,0,2=1;
b,1,a,0,0,0,2,1,1=2;	2,0,3,2,1,0,0,0,3=3;	b,2,1,1,c,0,1,3,a=a;	0,a,1,1,0,0,1,0,2=2;	2,0,2,0,1,0,0,0,0=1;
b,0,0,0,c,1,1,1,a=c;	1,3,2,0,1,0,0,0,2=3;	b,a,1,1,c,0,1,3,2=2;	a,0,b,1,2,1,0,0,0=b;	1,2,1,0,0,0,2,0,0=3;
c,a,b,0,0,0,2,1,1=2;	3,0,0,0,2,3,3,1,0=0;	3,0,2,a,1,1,0,0,0=2;	2,b,1,1,0,0,1,0,a=a;	2,0,3,3,2,0,0,0,0=3;
b,1,0,0,0,0,2,1,a=2;	2,0,0,0,0,1,3,3,3=0;	2,0,b,1,a,1,3,0,0=b;	2,1,1,1,0,0,0,1,a=a;	2,3,3,0,0,0,0,0,2=3;
2,c,b,0,0,0,a,1,1=a;	3,0,3,2,3,0,3,0,1=0;	1,2,a,b,0,0,1,0,3=2;	c,0,3,1,1,a,b,0,0=2;	3,0,1,0,0,0,2,2,3=1;
b,0,1,0,c,2,1,1,a=c;	0,3,3,1,0,0,0,0,3=3;	0,0,2,2,1,1,0,0,0=2;	0,1,0,0,0,a,3,1,1=4;	3,0,0,1,3,2,2,0,0=0;
b,a,2,0,0,0,c,1,1=c;	3,3,2,0,1,0,0,3,3=1;	2,0,a,2,2,1,0,0,0=a;	3,1,0,0,a,1,3,1,1=a;	0,0,3,3,2,0,0,0,0=3;
b,0,1,0,2,c,1,1,a=2;	3,0,1,1,0,0,0,0,3=1;	a,0,b,1,2,2,2,0,0=b;	a,0,0,0,b,1,1,3,3=b;	2,3,3,0,0,0,0,0,0=3;
2,1,0,0,0,0,b,1,a=b;	3,1,0,1,3,0,0,0,0=0;	a,2,2,1,1,1,b,0,0=2;	3,1,3,a,1,0,0,1,1=1;	3,0,1,0,0,0,2,0,3=1;
0,0,0,0,0,0,2,a,1=2;	0,0,0,0,2,1,1,0,0=2;	2,b,1,1,c,0,2,2,a=a;	1,3,1,4,c,1,1,a,b=c;	3,0,0,1,3,2,0,0,0=0;
b,0,0,1,2,1,1,1,a=2;	0,0,0,0,2,1,0,1,0=2;	2,a,2,b,0,0,1,0,2=1;	a,1,4,0,b,1,1,1,1=b;	2,0,0,1,1,0,0,0,0=0;
b,2,c,0,0,0,a,1,1=a;	0,0,2,0,1,1,0,0,1=2;	2,0,a,1,1,1,0,0,0=a;	b,4,0,0,c,1,1,1,a=c;	2,0,1,1,0,0,1,0,2=0;
2,0,1,2,b,c,1,1,a=b;	0,1,2,1,1,1,0,0,0=2;	a,0,b,a,1,1,2,0,0=b;	1,b,1,0,4,b,1,a,3=4;	2,0,a,1,b,0,1,0,0=a;
a,1,2,0,0,0,b,1,2=b;	2,0,0,0,0,1,2,1,0=0;	a,c,1,1,2,0,1,a,b=b;	4,1,0,0,0,b,a,1,1=0;	b,a,1,1,c,0,0,1,2=2;
0,0,0,0,0,2,2,0,0=2;	2,1,0,1,1,1,0,0,0=0;	b,0,c,1,a,1,a,0,0=c;	3,0,b,4,b,1,a,0,0=b;	a,0,b,1,2,0,1,0,0=b;
2,0,0,0,0,0,a,2,2=0;	0,1,1,1,0,3,3,0,0=1;	b,0,c,b,1,1,a,0,0=c;	4,a,1,0,0,a,b,3,b=0;	2,c,1,1,a,0,0,1,b=b;
2,0,0,1,b,1,1,1,a=b;	0,1,1,1,1,1,0,0,2=3;	a,c,1,1,d,0,1,a,b=b;	b,b,4,0,c,1,1,a,3=c;	a,c,1,1,2,0,0,1,b=b;
a,0,1,0,b,c,1,1,2=b;	1,0,0,1,1,1,0,2,1=3;	2,0,a,1,1,1,3,0,0=a;	a,0,b,1,4,a,3,0,0=0;	1,0,1,0,0,0,0,0,0=4;
0,0,0,0,0,0,0,1,2=2;	1,3,1,0,0,0,1,0,3=3;	2,0,a,1,1,3,0,0,0=a;	1,c,1,0,0,0,4,a,b=4;	2,0,0,1,1,4,0,0,0=0;
a,0,0,1,b,1,1,1,2=b;	1,0,1,0,0,0,1,3,3=3;	3,0,a,1,1,3,0,0,0=a;	3,0,b,c,1,1,a,0,0=b;	4,0,1,0,0,0,0,0,0=0;
2,0,0,0,0,0,0,1,2=0;	3,1,3,1,1,1,0,0,0=0;	a,0,b,1,1,1,3,0,0=b;	a,0,0,0,b,1,3,0,0=b;	1,0,1,0,0,0,4,0,0=4;
0,1,0,0,0,0,2,a,1=2;	3,0,0,1,1,1,3,0,1=1;	b,0,2,1,a,1,a,0,0=3;	b,0,0,0,c,1,1,3,a=c;	0,0,a,b,1,4,0,0,0=1;
2,1,0,0,0,0,a,2,1=0;	0,3,0,0,0,0,2,1,3=3;	b,2,1,1,c,0,1,b,a=a;	a,0,b,1,4,0,0,0,0=0;	1,a,b,c,0,0,4,0,0=0;
0,1,0,0,0,0,2,1,1=2;	3,1,3,0,0,2,1,0,0=1;	b,0,3,c,1,1,a,0,0=3;	4,b,1,0,0,0,0,0,a=0;	2,0,a,2,1,4,0,0,0=0;
2,1,0,0,0,0,0,1,1=0;	2,3,0,0,0,0,0,1,1=0;	a,c,1,1,b,0,1,a,3=2;	0,0,0,0,a,3,0,0,0=1;	1,a,b,2,0,0,4,0,0=0;
0,2,2,0,0,0,2,1,1=2;	0,1,0,0,0,0,3,1,3=2;	3,0,a,1,b,1,b,0,0=a;	a,1,b,c,1,1,3,0,0=b;	

Table A-3: The 179 transition rules (including variables) for our novel automaton. [PS,N,NE,E,SE,S,SW,W,NW=NS]

Table A-3 shows the transition rules for our novel self-replicating automaton (subsection 3.4.3), including four variables associated with the data states, while Table A-4 lists the 330 additional rules required by the embarked program to write “LSL” in the interior of the loop (subsection 3.4.5).

Table A-5 shows the rules required by the first version of our membrane builder (subsection 3.5.1), while in Table A-6 we introduce the transition rules of the augmented version (subsection 4.3.3).

Note the considerable number of states required by this automaton (23, including the quiescent state), as well as the important increase in the number of rules compared to the first version (207 versus 49). We will see that, fortunately, the increase in complexity of the automaton is much greater than that of the hardware (subsection B.1.2), another indication of the poor correlation between cellular automata and digital hardware.

VAR a = {5,6,7,8,9};	6,0,0,0,0,0,5,5,9=9;	7,0,0,5,5,5,6,0,0=6;	9,5,5,5,6,6,0,0,6=6;	0,0,5,8,0,0,0,0,0=5;
VAR b = {5,6,7,8,9};	9,6,0,0,6,5,5,5,6=6;	6,0,6,5,5,5,5,5,8=8;	6,9,6,5,5,5,8,6,0=8;	6,0,6,5,5,5,7,0,0=7;
VAR c = {5,6,7,8,9};	9,0,0,0,0,5,5,5,6=6;	8,0,0,6,6,5,5,5,6=6;	6,6,5,5,9,0,0,0,6=9;	7,0,6,5,5,6,6,0,0=6;
VAR d = {5,6,7,8,9};	5,9,0,0,0,0,0,0,5=7;	0,0,0,0,7,6,7,0,0=5;	6,0,0,5,5,5,8,0,0=8;	6,7,5,5,0,0,0,0,6=0;
1,5,7,2,1,0,0,1,1=5;	0,0,0,0,0,0,5,5,9=9;	7,0,0,0,7,5,6,7,0=5;	6,0,6,5,9,0,0,0,0=9;	6,0,6,5,0,0,0,0,0=0;
7,0,0,0,2,5,1,5,5=2;	6,5,7,0,0,6,6,5,5=7;	7,0,0,0,0,0,5,6,7=5;	9,6,5,5,6,0,0,0,6=6;	7,0,6,5,5,5,6,0,0=6;
2,0,0,0,7,1,5,1,7=7;	9,0,0,0,0,0,7,5,6=6;	7,0,0,7,6,6,0,0,0=6;	6,0,0,0,5,5,8,0,0=8;	6,0,6,5,5,0,0,0,0=0;
1,2,7,5,1,0,0,0,5=5;	0,9,0,0,0,0,0,0,7=6;	8,0,6,5,5,5,5,5,6=6;	6,0,6,5,5,6,9,0,0=9;	6,0,6,5,5,5,0,0,0=0;
5,7,2,7,1,0,0,1,1=1;	7,6,9,0,0,0,0,0,5=0;	6,7,6,5,5,5,8,6,0=8;	9,0,6,5,6,0,0,0,0=6;	6,0,6,6,5,5,7,0,0=7;
7,0,0,0,5,5,1,1,2=5;	7,5,6,0,0,6,6,5,5=6;	6,5,5,5,5,5,8,0,6=8;	0,0,0,0,5,5,8,0,0=8;	6,0,0,0,6,5,7,0,0=7;
5,0,0,0,6,1,5,1,7=6;	6,5,7,0,6,5,5,5,5=7;	8,6,6,5,5,5,6,6,0=6;	6,0,6,5,5,5,9,0,0=9;	7,0,6,6,5,5,6,0,0=6;
1,5,6,6,1,0,0,0,5=6;	0,0,0,0,0,0,6,0,6=6;	0,0,0,0,5,6,0,0,0=5;	6,9,5,5,8,0,0,0,6=8;	7,0,0,0,6,5,6,0,0=6;
0,1,5,6,0,0,0,0,0=5;	7,5,6,0,6,5,5,5,5=6;	6,0,0,0,8,1,6,5,2=8;	9,0,6,5,5,6,6,0,0=6;	6,0,0,0,6,5,5,6,7=7;
0,5,6,1,0,0,0,0,5=6;	6,6,0,0,6,5,5,5,7=7;	6,0,0,0,8,6,5,1,2=8;	8,0,0,5,5,5,6,0,0=6;	7,0,0,0,6,6,0,0,0=0;
0,0,5,6,0,0,0,0,0=5;	6,6,6,0,0,0,0,0,0=5;	6,0,0,0,8,5,1,1,2=8;	8,0,0,0,5,5,6,0,0=6;	6,7,6,7,5,5,0,0,0=0;
0,5,6,0,0,0,0,0,5=6;	6,0,0,0,6,0,5,5,7=7;	6,5,5,5,8,6,0,0,0=8;	9,0,6,5,5,5,6,0,0=6;	7,0,0,0,6,5,5,6,6=6;
6,0,0,0,7,1,6,5,6=7;	7,6,0,0,6,5,5,5,6=6;	0,0,5,5,6,0,0,0,0=6;	5,0,0,0,0,0,5,6,8=8;	6,0,0,0,6,5,5,5,7=7;
6,6,7,6,1,0,6,5,5=7;	6,5,7,0,0,7,6,5,5=7;	8,5,5,5,5,5,6,0,6=6;	8,0,0,0,0,0,5,6,6=6;	6,0,0,0,6,5,0,0,0=0;
6,0,0,0,7,6,5,1,6=7;	6,0,0,0,6,5,0,5,7=7;	6,5,9,0,0,6,6,5,5=9;	0,8,0,0,0,0,0,5,5=5;	6,0,0,0,6,6,5,0,7=7;
7,0,0,0,6,1,6,5,6=6;	7,0,0,0,6,0,5,5,6=6;	6,0,5,5,8,0,0,0,0=8;	6,5,0,0,0,7,6,5,5=0;	6,0,0,0,0,0,6,5,7=7;
7,7,6,6,1,0,6,5,5=6;	7,0,0,0,6,5,0,5,6=6;	6,0,0,6,6,5,5,5,8=8;	6,5,0,0,6,5,5,5,5=0;	7,0,0,0,6,6,5,0,6=6;
6,5,7,1,0,0,6,5,5=7;	6,0,0,0,6,5,5,0,7=7;	6,6,0,0,6,5,5,5,9=9;	6,0,6,6,5,5,9,0,0=9;	7,0,0,0,0,0,6,5,6=6;
6,0,0,0,9,1,6,5,6=9;	6,5,8,0,0,7,6,5,5=8;	8,5,5,5,6,6,0,0,6=6;	6,0,0,0,6,5,9,0,0=9;	6,6,7,0,0,0,6,5,5=7;
7,5,6,1,0,0,6,5,5=6;	7,0,0,0,6,5,5,0,6=6;	6,0,0,0,6,0,5,5,9=9;	9,0,6,6,5,5,6,0,0=6;	7,6,6,0,0,0,6,5,5=6;
6,5,7,0,0,0,6,5,5=7;	6,0,0,0,0,5,5,5,7=7;	0,0,5,5,8,0,0,0,0=8;	6,0,0,0,6,5,5,5,0=0;	6,0,0,0,6,7,5,0,0=0;
7,5,6,0,0,0,6,5,5=6;	6,6,9,2,1,0,6,5,5=9;	6,5,8,0,0,9,6,5,5=8;	6,0,0,0,6,0,5,5,0=0;	6,0,0,0,0,0,6,5,0=0;
6,0,0,0,9,6,5,1,6=9;	9,0,0,0,2,1,6,5,6=2;	8,5,5,5,6,0,0,0,0=6;	6,0,0,6,6,5,5,5,7=7;	6,0,0,0,0,0,5,5,0=0;
6,6,9,6,1,0,6,5,5=9;	9,0,0,0,2,9,5,1,6=2;	5,0,0,0,5,6,8,0,0=8;	6,0,0,0,6,5,5,6,9=9;	6,0,0,0,0,5,5,5,0=0;
9,0,0,0,6,1,6,5,6=6;	2,0,0,0,6,1,9,5,9=6;	6,6,6,5,5,5,8,6,0=8;	9,0,0,0,6,5,6,0,0=6;	5,5,5,5,1,0,5,5,5=7;
9,0,0,0,6,9,5,1,6=6;	9,9,2,6,1,0,6,5,5=6;	6,0,0,0,6,5,0,5,9=9;	9,0,0,0,6,5,5,6,6=6;	5,0,0,0,2,1,7,5,5=2;
6,5,9,1,0,0,6,5,5=9;	2,0,0,0,6,6,5,1,9=6;	9,0,0,0,6,0,5,5,6=6;	6,0,0,0,6,5,5,5,9=9;	5,0,0,0,2,7,5,1,5=2;
9,5,6,1,0,0,6,5,5=6;	7,0,0,0,6,5,5,5,6=6;	8,0,5,5,6,0,0,0,0=6;	6,0,6,9,5,5,8,0,0=8;	2,0,0,0,5,1,7,5,5=5;
6,5,9,0,0,0,6,5,5=9;	6,0,0,0,0,0,5,5,7=7;	6,0,0,0,6,5,5,0,9=9;	6,0,0,0,6,5,0,5,0=0;	2,0,0,0,5,7,5,1,5=5;
9,5,6,0,0,0,6,5,5=6;	0,0,0,0,0,0,5,5,7=7;	8,0,0,0,5,6,6,0,0=6;	6,6,6,5,5,5,7,6,0=7;	a,0,0,0,b,1,7,5,c=b;
2,0,0,0,6,1,6,5,6=6;	5,7,0,0,0,0,0,0,5=7;	9,0,0,0,6,5,0,5,6=6;	6,0,0,0,6,5,5,0,0=0;	a,0,0,0,b,7,5,1,c=b;
2,0,0,0,6,6,5,1,6=6;	0,0,0,0,0,0,7,6,0=7;	0,0,0,0,0,5,5,8,0=5;	6,0,0,0,6,5,8,0,0=8;	7,0,0,0,2,1,7,5,5=2;
9,9,6,2,1,0,6,5,5=6;	7,0,0,0,0,5,5,5,6=6;	6,0,0,0,5,1,6,5,6=5;	6,0,0,0,0,5,5,5,9=9;	7,0,0,0,2,7,5,1,5=2;
6,0,0,0,2,1,9,5,9=2;	7,0,0,0,0,7,6,6,0=6;	6,5,5,5,8,6,0,0,6=8;	8,0,6,6,5,5,6,0,0=6;	2,0,0,0,7,1,7,5,7=7;
6,0,0,0,2,6,5,1,9=2;	6,0,0,7,6,5,5,5,7=7;	6,0,0,6,6,5,5,5,9=9;	9,0,0,0,6,5,5,5,6=6;	2,0,0,0,7,7,5,1,7=7;
6,5,7,0,0,0,0,5,5=7;	7,7,0,0,0,0,7,5,6=5;	9,0,0,0,6,5,5,0,6=6;	6,0,0,0,6,5,5,6,8=8;	7,0,0,0,5,1,7,5,2=5;
6,5,7,0,0,0,0,0,5=7;	7,6,7,0,0,0,0,0,5=5;	6,0,0,0,5,6,5,1,6=5;	8,0,0,0,6,5,6,0,0=6;	7,0,0,0,5,7,5,1,2=5;
5,0,5,7,0,0,0,0,0=7;	0,0,0,0,0,0,7,6,7=5;	6,6,5,5,1,0,6,5,5=5;	8,0,0,0,6,5,5,6,6=6;	6,0,0,0,2,1,7,5,9=2;
0,5,7,0,0,0,0,0,5=7;	0,0,0,0,0,0,5,6,0=5;	6,6,5,5,8,0,0,0,6=8;	7,6,6,5,5,5,6,6,0=6;	6,0,0,0,2,7,5,1,9=2;
0,6,0,0,0,0,0,0,7=7;	6,0,6,5,5,5,5,5,7=7;	9,0,0,6,6,5,5,5,6=6;	6,5,5,5,5,5,7,0,6=7;	2,0,0,0,6,1,7,5,6=6;
7,5,6,0,0,0,0,5,5=6;	7,0,0,6,6,5,5,5,6=6;	6,0,6,5,5,5,5,5,9=9;	6,0,0,0,6,5,5,5,8=8;	2,0,0,0,6,7,5,1,6=6;
6,5,9,0,0,7,6,5,5=9;	6,5,8,0,6,5,5,5,5=8;	6,5,5,1,0,0,6,5,5=5;	6,0,0,7,6,5,5,5,0=0;	9,0,0,0,2,1,7,5,6=2;
7,6,0,0,0,0,0,7,6=6;	8,5,6,0,0,6,6,5,5=6;	6,5,5,0,0,0,6,5,5=0;	6,0,6,5,5,5,5,5,0=0;	9,0,0,0,2,7,5,1,6=2;
7,0,5,6,7,0,0,0,0=5;	6,6,0,0,6,5,5,5,8=8;	5,0,5,5,6,6,5,0,0=0;	6,5,5,5,7,6,0,0,6=7;	2,0,0,0,6,1,7,5,9=6;
7,5,6,7,0,0,0,0,7=5;	8,5,6,0,6,5,5,5,5=6;	5,5,5,1,0,0,6,5,5=0;	8,0,0,0,6,5,5,5,6=6;	2,0,0,0,6,7,5,1,9=6;
0,6,7,0,0,0,0,0,7=5;	8,6,0,0,6,5,5,5,6=6;	5,1,5,5,5,6,5,0,0=0;	7,5,5,5,5,5,6,0,6=6;	6,0,0,0,8,1,7,5,2=8;
0,6,0,0,0,0,0,0,5=5;	6,0,0,0,6,0,5,5,8=8;	6,0,6,5,8,0,0,0,0=8;	6,6,5,5,7,0,0,0,6=7;	6,0,0,0,8,7,5,1,2=8;
9,5,6,0,0,6,6,5,5=6;	6,0,0,0,6,5,0,5,8=8;	6,6,6,5,5,5,9,6,0=9;	7,5,5,5,6,6,0,0,6=6;	6,0,0,0,7,5,1,1,6=7;
6,5,9,0,6,5,5,5,5=9;	8,0,0,0,6,0,5,5,6=6;	8,6,5,5,6,0,0,0,6=6;	6,7,6,5,5,5,0,0,0=0;	7,0,0,0,6,5,1,1,6=6;
6,0,0,0,8,1,6,5,6=8;	8,0,0,0,6,5,0,5,6=6;	9,0,6,5,5,5,5,5,6=6;	6,0,0,0,6,6,0,5,8=8;	6,0,0,0,9,5,1,1,6=9;
6,6,8,6,1,0,6,5,5=8;	7,0,6,5,5,5,5,5,6=6;	6,0,0,0,0,0,6,5,5=0;	6,0,6,5,7,0,0,0,0=7;	9,0,0,0,6,5,1,1,6=6;
6,0,0,0,8,6,5,1,6=8;	0,0,0,0,5,5,6,0,0=6;	6,0,6,5,5,6,8,0,0=8;	7,6,5,5,6,0,0,0,6=6;	6,0,0,0,2,5,1,1,9=2;
8,0,0,0,6,1,6,5,6=6;	6,0,0,5,5,5,7,6,0=7;	6,0,0,9,6,5,5,5,8=8;	6,5,5,5,5,5,0,0,6=0;	2,0,0,0,6,5,1,1,6=6;
0,0,0,0,0,0,5,5,6=6;	6,0,0,0,6,5,5,0,8=8;	6,5,5,5,5,5,9,0,6=9;	6,0,0,0,0,0,5,0,8=8;	6,0,0,0,8,5,1,1,6=8;
6,6,0,0,0,5,5,5,9=9;	6,5,8,0,0,6,6,5,5=8;	9,6,6,5,5,5,6,6,0=6;	6,0,6,5,5,6,7,0,0=7;	8,0,0,0,6,5,1,1,6=6;
9,5,6,0,6,5,5,5,5=6;	6,0,0,7,6,5,5,5,8=8;	8,0,6,5,6,0,0,0,0=6;	7,0,6,5,6,0,0,0,0=6;	9,0,0,0,2,5,1,1,6=2;

Table A-4: The 330 additional rules required for the embarked program.
[PS,N,NE,E,SE,S,SW,W,NW=NS]

6,5,8,1,0,0,6,5,5=8;	6,0,0,0,5,5,7,0,0=7;	6,5,0,0,0,0,6,5,5=0;	6,5,5,5,0,0,0,0,6=0;	2,0,0,0,6,5,1,1,9=6;
8,0,0,0,6,8,5,1,6=6;	7,0,6,5,5,5,6,6,0=6;	6,5,5,5,9,6,0,0,6=9;	0,0,0,0,0,0,5,5,8=8;	6,0,0,0,5,5,1,1,6=5;
8,8,6,6,1,0,6,5,5=6;	8,0,0,0,6,5,5,0,6=6;	6,0,6,5,5,5,8,0,0=8;	8,0,0,0,0,5,5,0,6=6;	5,0,0,0,2,5,1,1,5=2;
6,5,8,0,0,0,6,5,5=8;	0,0,0,0,5,5,7,0,0=7;	8,0,6,5,5,6,6,0,0=6;	5,6,8,0,0,0,0,0,5=8;	2,0,0,0,5,5,1,1,5=5;
8,5,6,1,0,0,6,5,5=6;	5,0,0,0,0,0,5,7,0=7;	9,5,5,5,5,5,6,0,6=6;	8,0,0,0,0,0,5,5,6=6;	7,0,0,0,2,5,1,1,5=2;
8,5,6,0,0,0,6,5,5=6;	0,0,0,0,7,6,0,0,0=7;	8,0,6,5,5,5,6,0,0=6;	8,6,6,0,0,0,0,0,5=6;	2,0,0,0,7,5,1,1,7=7;

Table A-4: The 330 additional rules required for the embarked program. [PS,N,NE,E,SE,S,SW,W,NW=NS]

2,0,0,0,4=6;	6,0,0,0,5=5;	2,0,0,4,0=7;	5,4,0,4,0=4;	4,4,4,5,0=5;	4,7,6,0,5=5;	5,4,4,0,0=4;
4,0,4,0,5=5;	5,0,6,0,4=4;	7,0,0,4,0=4;	5,7,0,4,0=4;	5,4,4,4,0=4;	4,7,6,5,0=5;	4,1,4,0,5=5;
5,0,4,0,4=4;	2,0,0,0,5=3;	0,0,0,7,0=2;	7,0,0,5,0=5;	0,0,0,7,6=2;	4,7,6,5,5=5;	5,1,4,0,4=4;
6,0,0,0,4=4;	5,0,2,0,4=4;	4,4,4,0,5=5;	5,2,0,4,0=4;	2,0,0,5,5=3;	4,5,1,4,0=5;	4,0,4,0,0=0;
0,0,0,0,6=2;	3,0,0,0,4=4;	5,4,4,0,4=4;	2,0,0,5,0=3;	3,0,0,4,4=4;	5,4,1,4,0=4	4,1,4,0,0=0;
4,0,6,0,5=5;	0,0,0,3,0=2;	4,7,0,5,0=5;	4,4,0,5,0=5;	4,4,4,5,5=5;	4,0,1,4,0=0;	1,0,0,1,4=0;
4,0,2,0,5=5;	0,0,0,0,3=2;	4,2,0,5,0=5;	3,0,0,4,0=4;	5,4,4,4,4=4;	4,5,4,0,0=5;	1,0,0,4,4=0;

Table A-5: The 49 transition rules of the original membrane builder. [PS,N,E,S,W=NS;]

A.3 Technical Issues

CAEditor consists of approximately five thousand lines of C code, and requires the UNIX-based graphic library `libsx` (developed by Dominic Giampaolo) to operate.

The version currently available for distribution has a few relatively important faults:

- It is not cross-platform, requiring a UNIX workstation to operate.
- The `libsx` library, while easy to use, was designed for much smaller programs, and is at times unwieldy.
- It requires a considerable amount of computer memory particularly when handling large cellular spaces.
- It requires an explicit definition of the transition rules. That is, it does not allow the behavior of the automaton to be described as regular expressions or mathematical equations (as was the case for Life, the simple automaton introduced in section 3.1).

It is our intention to design a second version of our tool (indeed, such a process has already started, only to be interrupted by other concerns) in order to address some, if not all, of these problems.

On the other hand, the available version is in fairly good working order (that is, it is relatively bug-free), and has been compiled and run without undue problems on a variety of machines, from Sun workstations to PCs running Linux. It can be obtained (bundled with the `libsx` library and a few basic instructions) from our anonymous FTP server at the address:

`ftp://lslsun.epfl.ch/pub/CAeditor/editor.tar.gz`

4,5,4,0,0=5;	0,0,0,0,21=5;	12,8,0,13,0=13;	7,4,18,12,0=8;	15,0,22,0,5=5;
5,6,1,4,0=6;	17,0,21,0,10=18;	16,0,20,0,17=17;	10,4,17,0,19=7;	7,14,18,4,0=3;
6,5,1,5,0=5;	4,4,0,7,0=11;	17,0,16,0,22=18;	19,0,10,0,16=20;	19,0,10,0,5=6;
5,6,1,5,0=6;	7,4,18,0,4=8;	22,0,17,0,17=21;	7,4,18,0,20=8;	7,14,18,0,6=3;
6,3,1,5,0=3;	18,0,17,0,7=15;	9,13,17,0,6=10;	8,12,16,0,21=9;	7,4,18,4,0=3;
3,4,1,6,0=4;	18,0,21,0,7=15;	14,13,0,13,0=13;	8,12,16,13,0=9;	11,4,0,3,0=4;
4,5,1,4,0=5;	21,0,5,0,18=22;	13,8,0,14,0=14;	4,0,0,12,20=8;	11,4,0,4,0=4;
3,0,1,6,0=0;	5,0,0,0,21=17;	8,12,0,13,0=9;	9,12,16,14,0=10;	11,0,0,4,0=4;
3,0,0,0,4=8;	4,4,0,11,0=11;	20,0,8,0,17=21;	9,12,16,0,22=10;	7,14,18,4,6=3;
0,0,0,8,0=4;	11,4,0,8,0=12;	17,0,20,0,18=18;	10,13,17,13,0=9;	7,4,18,4,6=3;
5,6,4,0,0=6;	15,0,22,0,8=16;	18,0,17,0,21=17;	9,13,17,14,0=10;	0,0,0,11,19=3;
4,1,4,0,5=5;	22,0,17,0,15=19;	10,13,17,0,3=7;	10,13,17,11,0=7;	3,0,0,12,20=8;
4,0,0,8,0=12;	17,0,0,0,22=18;	9,12,0,14,0=10;	7,14,18,12,0=8;	7,4,18,0,6=3;
4,0,8,0,5=5;	0,0,0,0,17=5;	14,9,0,13,0=13;	10,13,17,0,21=9;	15,0,6,0,5=5;
6,5,5,0,0=5;	5,0,0,0,18=18;	13,13,0,10,0=14;	9,13,17,0,22=10;	19,0,0,0,5=6;
5,1,4,0,6=6;	18,0,5,0,19=15;	7,14,18,0,4=8;	10,13,17,0,19=7;	0,0,0,0,19=3;
0,0,0,12,0=4;	19,0,18,0,16=20;	10,4,5,13,0=9;	7,14,18,0,20=8;	3,0,0,0,20=8;
6,1,5,0,5=5;	4,0,0,11,0=11;	13,13,0,14,0=14;	8,11,16,0,21=9;	6,1,5,3,1=3;
5,0,8,0,6=6;	11,4,0,12,0=12;	14,13,0,7,0=11;	9,11,16,0,22=10;	3,6,0,4,1=4;
8,12,0,0,5=9;	8,12,16,0,5=9;	21,0,17,0,18=22;	10,11,17,0,21=9;	4,4,0,5,1=5;
4,0,0,12,0=12;	11,0,0,12,0=12;	18,0,21,0,17=17;	8,12,0,13,21=9;	5,4,0,6,1=6;
12,4,0,9,0=4;	0,0,0,11,0=3;	21,0,8,0,18=22;	9,12,0,14,22=10;	6,5,0,5,1=5;
0,0,0,0,9=5;	18,0,0,0,15=15;	8,12,0,0,21=9;	10,4,5,13,21=9;	5,5,0,6,1=6;
9,12,0,0,6=10;	15,0,18,0,20=16;	9,12,0,0,22=10;	9,4,17,14,22=10;	6,5,0,3,1=3;
6,0,9,0,5=5;	3,0,0,12,0=8;	22,0,9,0,17=21;	10,4,17,11,19=7;	3,6,0,0,1=0;
5,6,5,0,0=6;	9,12,16,0,6=10;	17,0,17,0,22=18;	7,4,18,12,20=8;	3,1,6,4,1=4;
12,0,0,4,0=4;	12,12,0,9,0=13;	13,9,0,14,0=14;	8,12,16,13,21=9;	3,1,5,0,4=4;
5,0,0,0,10=18;	16,0,20,0,9=17;	14,13,0,11,0=11;	9,12,16,14,22=10;	5,1,1,5,6=6;
10,4,5,0,5=9;	15,0,6,0,16=16;	11,14,0,8,0=12;	10,13,17,13,21=9;	6,1,1,5,3=3;
5,1,5,0,6=6;	6,0,0,0,15=19;	9,4,17,14,0=10;	9,13,17,14,22=10;	3,1,1,6,4=4;
6,3,5,0,0=3;	19,0,0,0,16=20;	14,9,0,11,0=11;	10,13,17,11,19=7;	4,1,4,5,1=5;
3,4,6,0,0=4;	20,0,16,0,17=21;	11,14,0,12,0=12;	7,14,18,12,20=8;	5,1,4,6,1=6;
6,1,5,0,3=3;	17,0,20,0,10=18;	17,0,21,0,18=18;	3,0,6,0,0=0;	4,1,1,4,5=5;
5,0,9,0,6=6;	10,13,17,0,5=9;	18,0,17,0,19=15;	3,1,6,0,0=0;	6,1,5,5,1=5;
18,0,0,0,9=17;	13,12,0,10,0=14;	10,4,5,0,21=9;	3,0,10,0,0=0;	5,1,1,4,6=6;
0,0,0,0,18=6;	12,12,0,13,0=13;	11,10,0,12,0=12;	7,14,18,0,0=3;	5,1,5,6,1=6;
3,1,6,0,4=4;	4,0,0,0,20=8;	10,4,17,11,0=7;	11,14,0,3,0=4;	6,1,1,5,5=5;
6,0,9,0,3=3;	16,0,16,0,21=17;	18,0,21,0,15=15;	15,0,22,0,3=5;	3,1,6,0,1=0;
9,4,17,0,6=10;	21,0,16,0,18=22;	21,0,9,0,18=22;	11,14,0,4,0=4;	3,1,1,6,0=0;
6,0,0,0,17=21;	18,0,21,0,9=17;	15,0,22,0,16=16;	19,0,18,0,5=6;	
3,0,10,0,4=4;	14,13,0,9,0=13;	22,0,9,0,15=19;	15,0,18,0,6=5;	
10,4,17,0,3=7;	13,12,0,14,0=14;	9,4,17,0,22=10;	11,10,0,4,0=4;	

Table A-6: The 207 transition rules of the augmented membrane builder. [PS,N,E,S,W=NS;]

APPENDIX B

A HARDWARE IMPLEMENTATION

It is a tradition of our laboratory to implement our designs as actual hardware prototypes, rather than limiting ourselves to software simulations. In many cases, this approach has led us to identify problems and possible improvements which were not apparent in simulation. Such was the case, for example, for the MicTree prototype described in subsection 2.2.4, which revealed the limitation of fixed-size cells. To a lesser extent, designing a prototype of our FPGA also revealed some ways to improve on our self-replication and self-repair mechanisms.

In this appendix, we will describe the logic layout of our MuxTreeSR (for *MuxTree with self-repair*) prototype (section B.1), and then cursorily describe the MuxNet (for *network of MuxTreeSR elements*) circuit (section B.2), an attempt to implement a 4x4 array of MuxTreeSR elements into a single commercial FPGA.

B.1 MuxTreeSR

In this section, we will describe in detail our prototype of the FPGA we developed in this thesis. After a brief overview (subsection B.1.1) of the system, we will present the logic layout of our circuit following a top-down approach: starting from the cellular automaton used for replication (subsection B.1.2), through the logic required for rerouting the array in the presence of faulty elements (subsection B.1.3) and the state machine which control the array's operation (subsection B.1.4), to the actual core of the MuxTreeSR element, consisting, as we have seen, of a programmable function (subsection B.1.5), a switch block for long-distance connections (subsection B.1.6), and a configuration register (subsection B.1.7).

B.1.1 Overview

Our laboratory's teaching activities include a number of laboratory sessions aimed at introducing the principles of logic design. In order to accomplish this task, we use a set of *logidules* (logic modules), plastic cubes which contain standard logic circuits (AND gates, RAMs, etc.). These modules can be connected together (not unlike a puzzle), automatically providing the circuits' power supply as well as the minimal connections between modules.

Aside from being an invaluable teaching tool, the logidules are also an interesting platform for the development of prototypes, as they can be easily exploited for the support circuitry which invariably surrounds any prototype. It is therefore not surprising that we decided to exploit the same approach in the design of the prototypes related to the Embryonics project, which we call Biodules (for *biologically-inspired logic modules*). The prototype of MuxTreeSR, the FPGA presented in this thesis, is no exception, and was embedded in a module called Biodule 603 (Fig. B-1).

Each Biodule 603 is designed to contain a single MuxTreeSR element, along with four copies of the cellular automaton used to define the blocks' membranes (the four copies were required in order to be able to use all the Biodules, as explained below in subsection B.1.2). The Biodule 603's displays include:

- four 7-segment displays (in the four corners) which show the state of the corresponding CA elements;
- a set of five 7-segment displays which show the value of the element's configuration;
- a set of four LEDs (top and bottom) used to indicate the source of the signals (in case of reconfiguration);
- two sets of LEDs which show the value of the functional outputs and the input and output values of the flip-flops of each of the two copies of the element's programmable function;
- a single LED which lights up whenever a fault is detected in the element;
- a three-color LED which denotes the state of the element (green for active, red for dead, yellow for spare or during the configuration and repair processes).

In addition to the displays, each Biodule 603 also contains a rotary encoder used to select one of ten possible faults to introduce in the element, and a push-button which activates the selected fault: a single push will introduce a temporary fault (which will disappear whenever the FPGA is reset), while a double push will introduce a permanent fault.

Invisible to the user, the Biodule 603 contains a small printed circuit board on which all the required circuitry is mounted. Aside from the displays mentioned above and the components (such as resistor nets) required for their control, the only "active" circuits are a Xilinx XC4013PQ240-4 FPGA [111], used to implement the MuxTreeSR element, and a 256k-bit EEPROM used to store the Xilinx's configuration bitstream.

Using a Xilinx FPGA with a locally-stored configuration to implement our elements allows us the opportunity of upgrading our design at a latter date without needing to build a different set of Biodules. In fact, the amount of programmable logic offered by the Xilinx chip is much greater than that required to implement our MuxTreeSR elements: our design uses approximately 15% of the

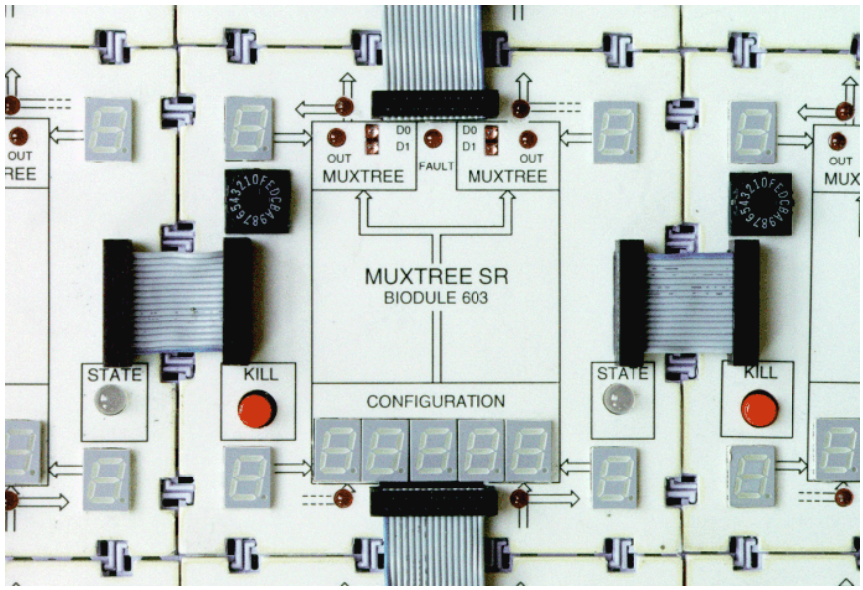

Figure B-1: A single element of the FPGA embedded into a Biodule 603. [Photo by André Badertscher]

available resources, a figure which includes all the circuitry required to control the displays. On the other hand, the XC4013 is the smallest Xilinx circuit available with enough I/O pins to satisfy our requirements: our design occupies only a fraction of the programmable logic, but uses 100% of the available pins.

Like all logidules, the Biodules 603 can be joined together to form a two-dimensional array (Fig. B-2), with the connections implemented either through the automatic contacts on the perimeter of the box (8 per side) or through four 16-bit-wide connectors. As of now, 18 modules are available, allowing us to build a 6x3 array of MuxTreeSR elements.

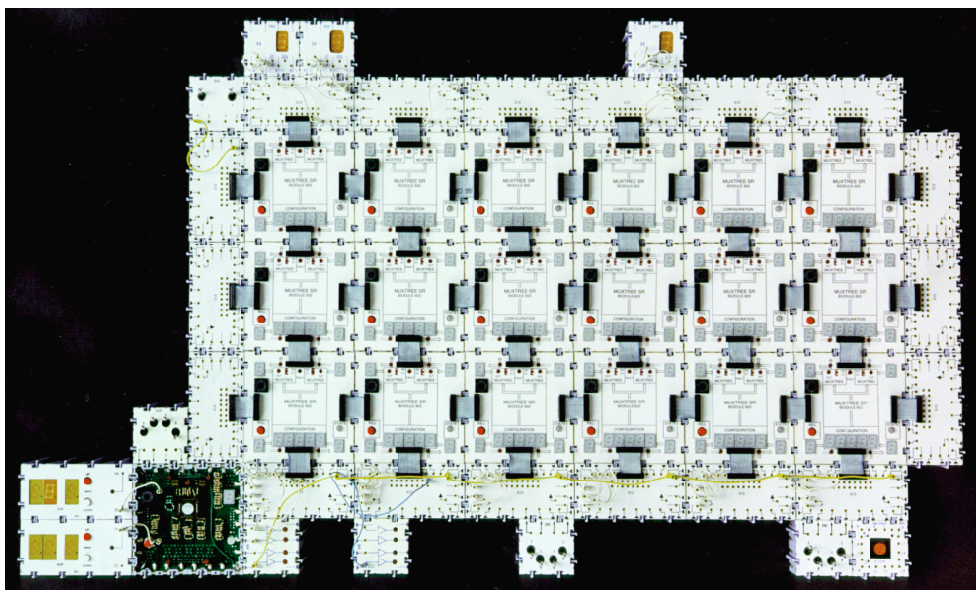

Figure B-2: The prototype: a 6x3 array of Biodules 603. [Photo by André Badertscher]

To operate the array, the following signals must be provided externally:

- a configuration bitstream, including the input sequence for the cellular automaton, the test pattern, and the blocks' configuration;
- two clocks, a faster one for the configuration, and a slower one for the operation of the array;
- a functional reset, which resets the elements' flip-flops to the default value defined in their configuration;
- a global reset, which resets the current configuration of the MuxTreeSR array;
- a power reset, which resets the configuration of the Xilinx chips.

B.1.2 The CA Level

The topmost level in the logic layout of our element is, of course, the interface between the I/Os of our element and the pins of the Xilinx chip. With a few exceptions (e.g., the encoding/decoding of the signals controlling the 7-segment displays), this relationship is one-to-one (i.e., a pin for each I/O line), and is therefore of no particular interest.

The topmost *relevant* level in our layout is therefore the cellular automaton which defines the size of our blocks (Fig. B-3). As we have seen, the CA elements are placed outside of the MuxTreeSR elements (Fig. 3-29), and in theory a single CA element per Biodule would be sufficient to realize our system. However, since each Biodule contains a single MuxTreeSR element, and since the CA membrane must completely surround the active logic elements, we were forced, in order to avoid "wasted" modules, to introduce *four* CA elements into each Biodule. This redundancy, which is not crucial since we use only a fraction of the Xilinx's programmable logic, allows us to obtain an usable logic element for every available Biodule.

Our first implementation of the cellular automaton was functionally equivalent to the automaton described in section 3.5, which allowed a fairly trivial transition to hardware. We soon realized, however, that the limitations imposed by this automaton (notably, the requirement that blocks be perfect squares) were too restrictive, and we improved its functionality to that described in subsection 4.3.3. The new automaton allows both the definition of rectangular blocks and the assignment of spare columns of elements. Unfortunately, this improved functionality comes at the cost of augmented complexity, and the hardware implementation of the new automaton is far from trivial.

As we have seen in section A.2, the new automaton uses 23 states and a neighborhood of 5, with a total of 207 transition rules. A trivial implementation of this automaton would require a look-up table of 23^5 5-bit words, a 5-bit register to hold the present state, and eight sets of 5-bit-wide connections with the neighboring elements (four input and four output busses). Even if it were possible to reduce the size of the look-up table (since the automaton uses a fraction of all pos-

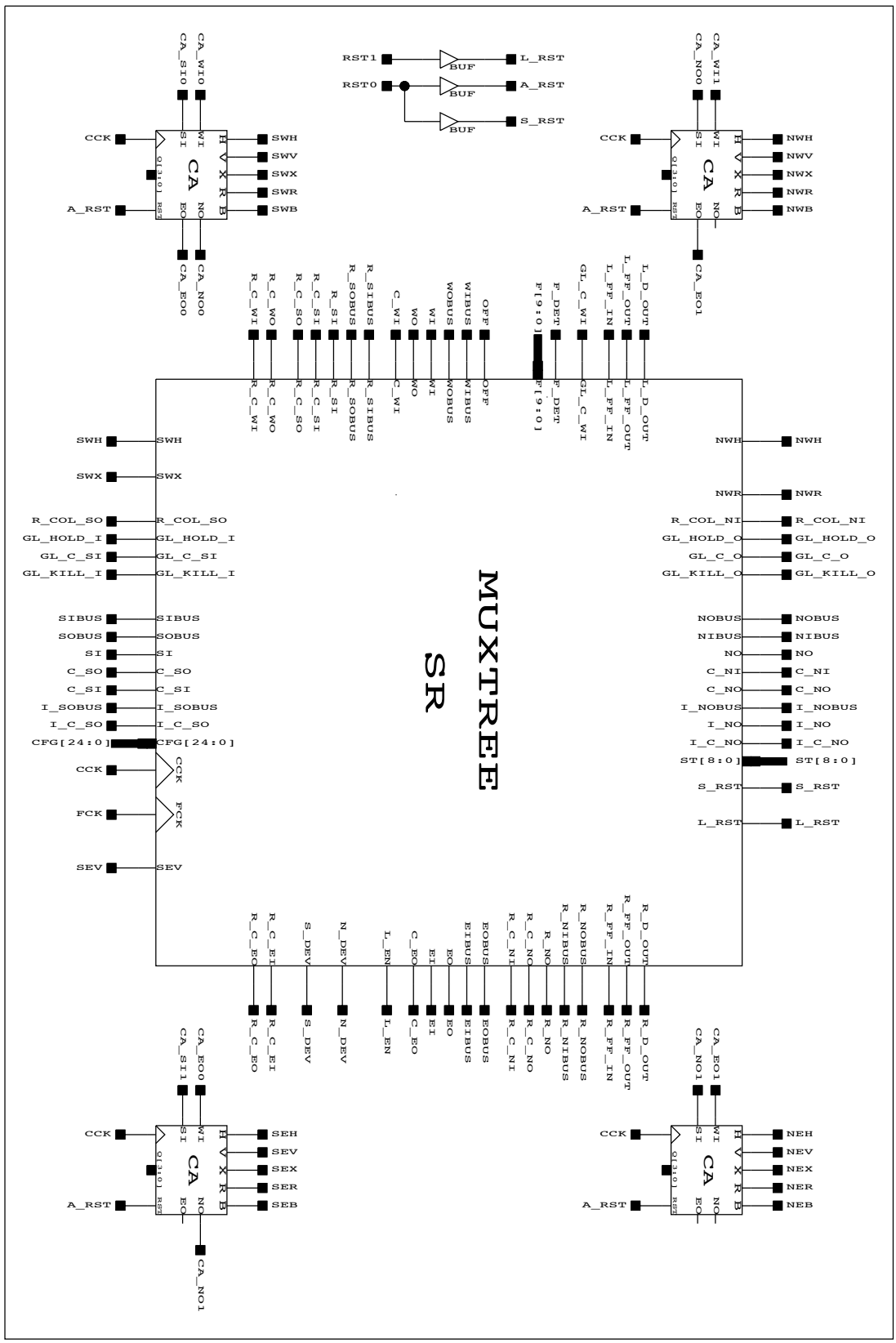

Figure B-3: The cellular automaton level.

sible transition rules), such an approach is obviously out of the question, if not for the size of the register (a 5-bit register might be barely acceptable), at least for the size of the busses.

Fortunately, it is possible to enormously simplify the layout of the elements through two techniques:

1) serializing the connections, which reduces the size of the busses to a single line, and
2) hard-wiring the transition rules, which does away with the need for a look-up table.

These techniques, while not applicable to all cellular automata, are extremely useful for our automaton which, as we mentioned, was designed for exactly this kind of hardware implementation.

The resulting circuit (Fig. B-4) consists of four D-type flip-flops, a few logic gates, and four 1-bit-wide connections (inputs from the south and west neighbors, outputs to the north and east neighbors). Its operation, because of the hard-wired transitions, is not apparent at first glance, and requires some explaining.

The configuration stream for the automaton (Fig. 3-29) is stored in an EEPROM (Electrically-Erasable Programmable Read-Only Memory) outside of the array, and enters the array from the lower left corner element. The four possible states of the automaton are encoded using three bits:

State	Encoding
Horizontal wall	`001`
Spare column	`011`
Vertical wall	`101`
Junction	`111`

Table B-1: Encoding of the states of the automaton.

The states enter the array left-bit-first, so that the first bit of each state is always a `1`. Each element receives information indifferently from the south or the west inputs. As long as the element receives only `0`s, nothing happens, and the automaton remains in the quiescent state `0000`. As the first `1` arrives, it is stored in the first flip-flop (`Q3`), and shifted from there to the second (`Q2`) and the third (`Q1`). At this point, when `Q1=1` (since the first bit of an incoming state is always `1`) and `Q0=0` (since the element has not yet been configured), the automaton makes a decision:

- if `Q3=0` (that is, if the incoming state is either a horizontal wall or a spare column) and the data is coming in from the south and not from the west, then the incoming state is ignored and the flip-flops reset, since horizontal walls and spare columns should only propagate horizontally;

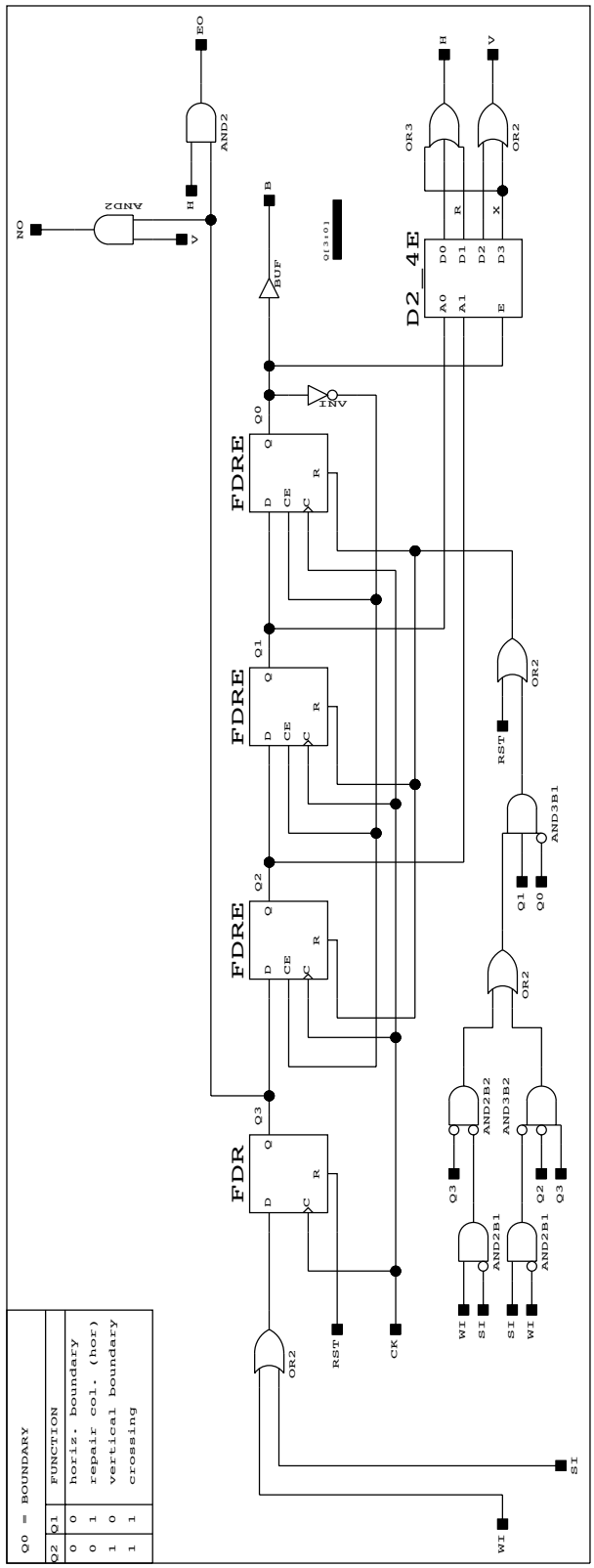

Figure B-4: The layout of a single CA element.

- if Q3=1 and Q2=0 (that is, if the incoming state is a vertical wall) and the data is coming in from the west and not from the south, then the incoming state is ignored and the flip-flops reset, since vertical walls should only propagate vertically;
- in all other cases, the flip-flops are not reset and the incoming state is allowed to shift once more, so that Q0=1 and Q1 and Q2 uniquely identify the state of the element.

Once the element is configured (Q0=1), the shift chain is broken: Q2, Q1, and Q0 remain fixed, thus setting the state of the element, while the incoming stream passes through Q3 and is propagated to the north (if the state is a vertical wall or a junction) and/or to the east (if the state is a horizontal wall, a spare column or a junction). The state itself is then decoded to provide the required signals which are used to redirect the flow of the FPGA's configuration.

This small circuit allows the user to very simply define rectangular membranes of any given size and to program the frequency and placement of the spare columns, and is thus functionally equivalent to the automaton described in subsection 4.3.3.

B.1.3 The Self-Repair Level

Immediately below the cellular automaton level we find a level dedicated to the rerouting of the connections for self-repair (Fig. B-5).

As we have seen in chapter 4, our self-repair mechanism reconfigures the array so as to avoid using the faulty elements, and reroutes the connections around them. The minimal set of connections that need to be rerouted to implement the pattern shown in Fig. 4-17 consists of:

- on the north side, the input and output long-distance busses NIBUS and NOBUS, the short-distance output NO, and the input and output configuration lines C_NI and C_NO (that is, the lines used to propagate the configuration through all the elements in a block);
- on the south side, the input and output long-distance busses SIBUS and SOBUS, the short-distance input SI, and the input and output configuration lines C_SI and C_SO;
- on the west side, if the element is active the output long-distance bus WOBUS receives the corresponding output of the internal switch block and the short-distance output WO receives the short-distance input SI, while if the element is dead, WOBUS receives the input long distance bus EIBUS and WO receives the short-distance input EI;
- on the east side, if the element is active the output long-distance bus EOBUS receives the corresponding output of the internal switch block, the short-distance output EO receives the short-distance input SI, and the output configuration line C_EO receives the signal generated within the element, while if the element is dead, EOBUS

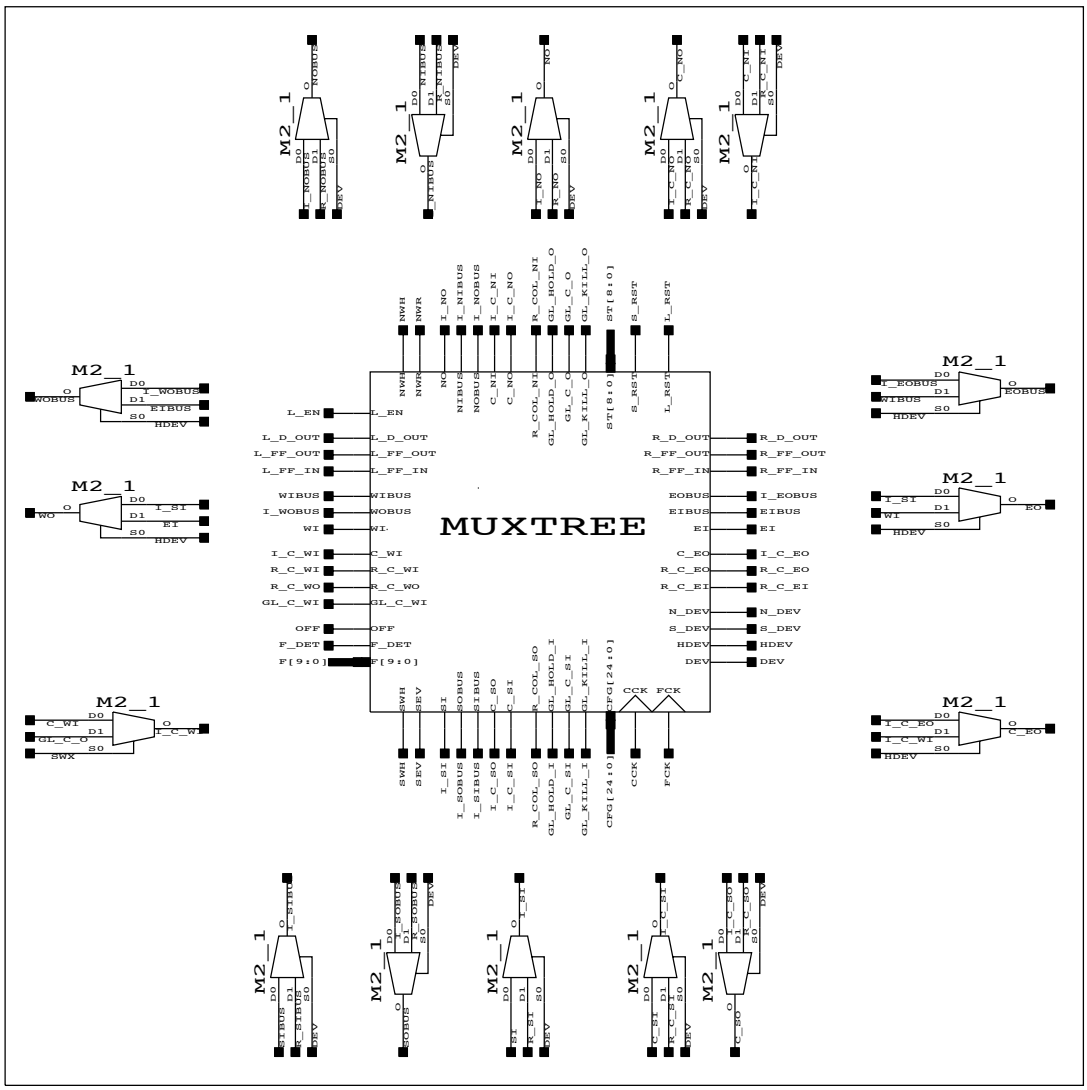

Figure B-5: The rerouting logic.

receives the input long distance bus WIBUS, EO receives the short-distance input WI, and C_EO receives the input configuration line I_C_WI;

In addition, we put at this level one extra multiplexer whose function is to connect the global configuration line GL_C_O to the west input configuration line I_C_WI in entry point elements. Placing the multiplexer at this level is not strictly necessary, but simplifies the layout.

B.1.4 The Control Logic

Figure B-6 shows the core of the MuxTreeSR element. The three main subcircuits described in chapter 4 (the programmable function, the switch block, and the configuration register) are clearly visible. Also visible is a fourth subcircuit, labeled STATE, which has not so far been described.

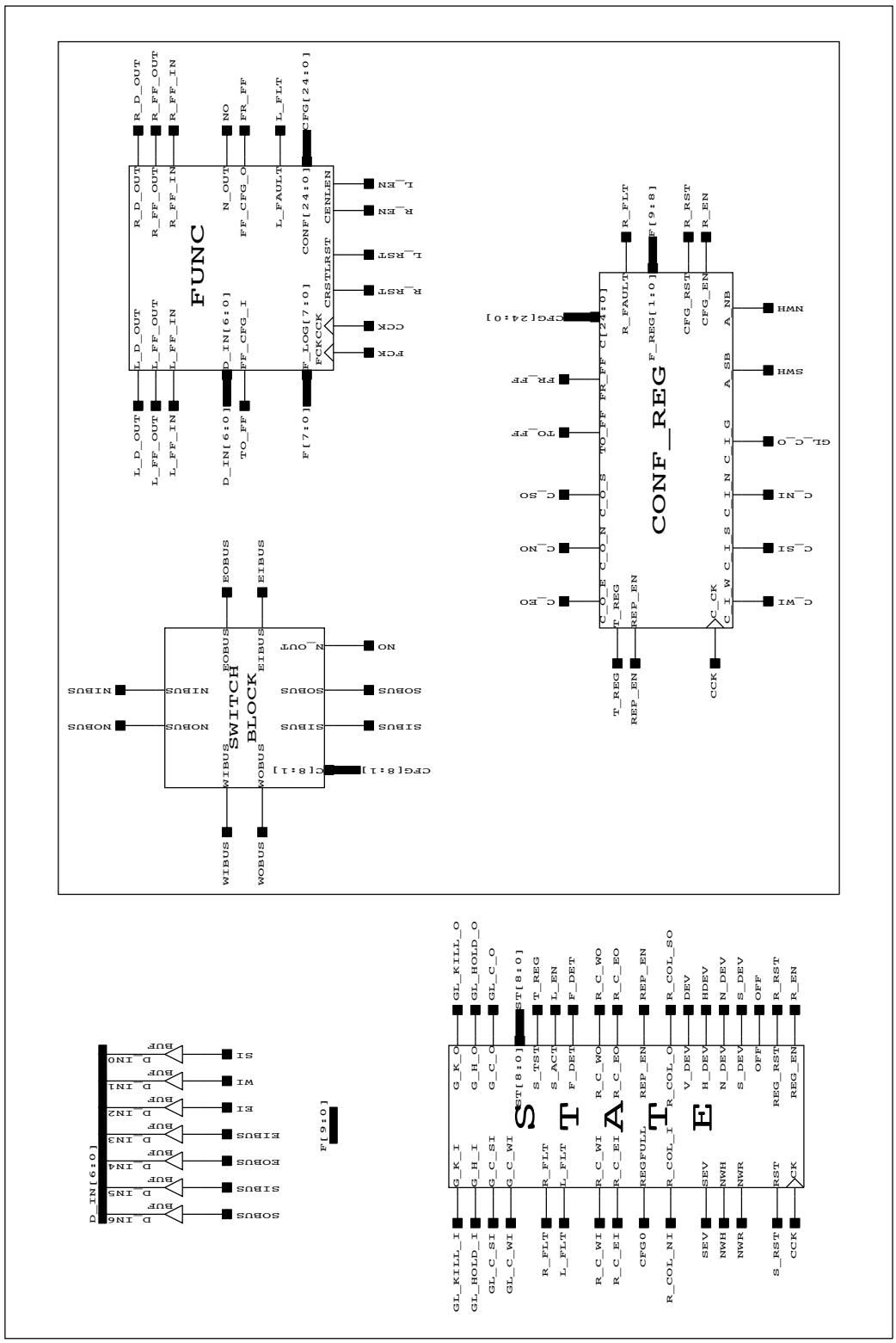

Figure B-6: The core of the MuxTreeSR element.

In this subcircuit (Fig. B-7), we collected all of the control logic required to handle all the phases of the operation of our array. Its size, while consequent, is not quite as important as it might appear, since it also includes a considerable amount of logic required for the operation of the Biodule (logic which would then disappear in a VLSI implementation). Moreover, some of the signals generated in this subcircuit are required not by the system itself, but rather by the use of a Xilinx FPGA, which is somewhat limited in its versatility and has some peculiar requirements which complicate the design of our control system (for example, it has difficulty handling bidirectional busses crossing multiple chips, a non-trivial problem where our global control lines are concerned).

The heart of the control logic is a *state machine*, implementing the state graph of Fig. B-8. Each element of the array can be in one of 7 possible states, and the transitions are controlled by two signals:

- A global hold signal `G_H_I`. This signal is used to "freeze" the execution of the circuit. It can be seen as a single global line which is usually pulled to `0` by a pull-down resistor outside the array. Any element can pull it to `1` by setting the `G_H_O` signal. It is set to `1` while the register is being tested, while the configuration bitstream is propagating, and while the reconfiguration mechanism is active.
- An internal fault detection signal `F_SPOT`. This signal is set to `1` if and only if a fault is detected inside the element itself.

The seven states operate as follows:

- `S_TST` (`000`) is the initial state. In this state, the element is waiting for the test pattern (subsection 4.2.5) to come in and check the register, and imposing a `1` on `G_H_O`. If a fault is spotted during this test, the element immediately dies (`DEAD`). If, on the other hand, no fault is spotted, the global hold signal is released. As soon as all the elements have released the hold signal (`G_H_I=0`), the element starts to wait for the configuration to arrive (`S_CFG`).
- `S_DIE` (`001`) is the state acquired by the faulty element while the repair process is occurring. As soon as the reconfiguration is complete (`G_H_I=0`), the element dies (`DEAD`).
- `S_FLT` (`010`) is a temporary state, introducing a one-clock-cycle delay between the detection of a fault and the beginning of the repair process. It is not in fact required for the operation of the array, but is a consequence of certain limitations of the Xilinx chip, and should disappear in a VLSI implementation.
- `S_REP` (`100`) is the state acquired by all non-faulty elements during the repair process. The elements go back to the active state (`S_ACT`) as soon as the reconfiguration is finished.
- `S_CFG` (`101`) is the configuration state. Elements remain in this state while they are waiting for the array to be configured. In this state,

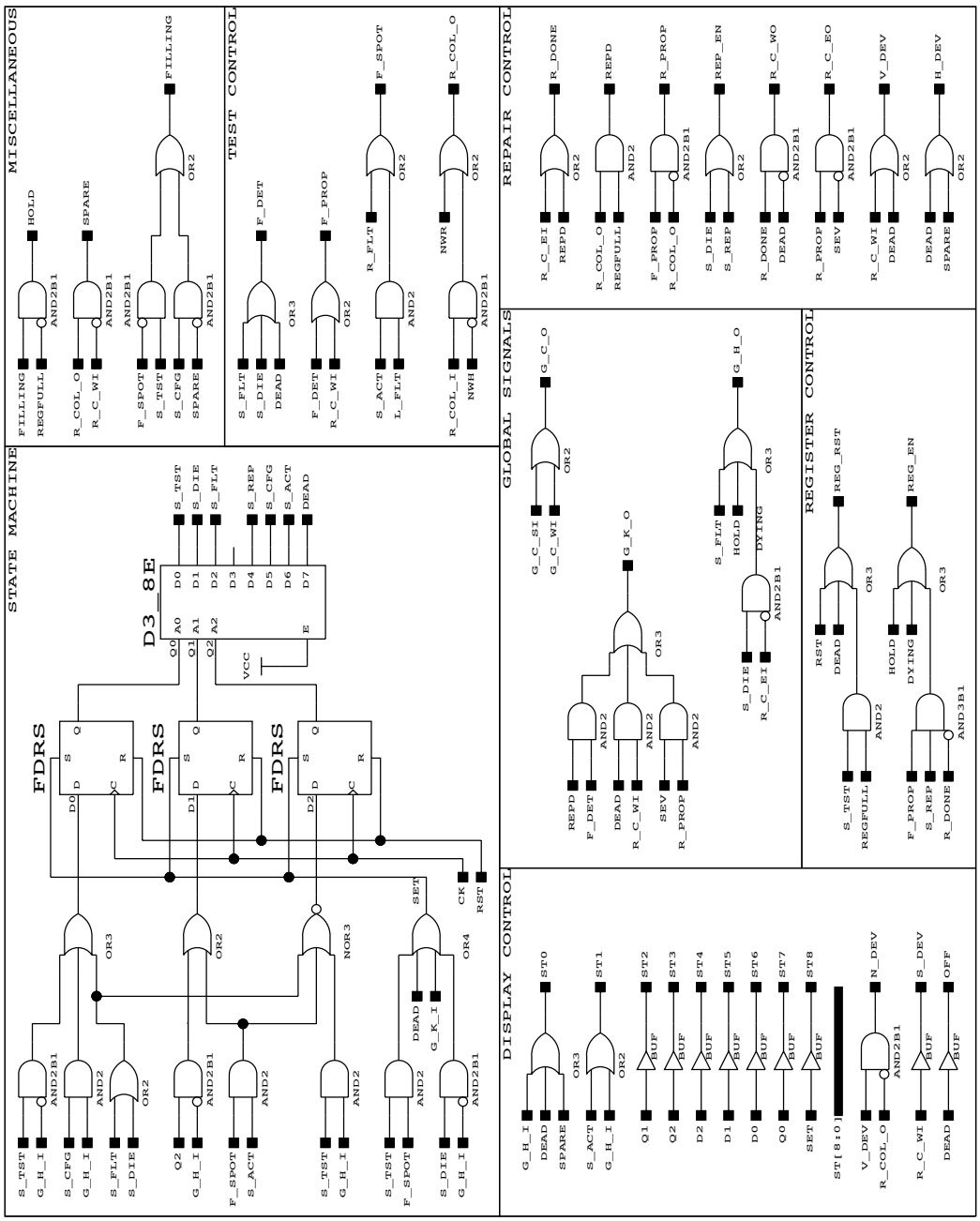

Figure B-7: The state machine and control logic for the element.

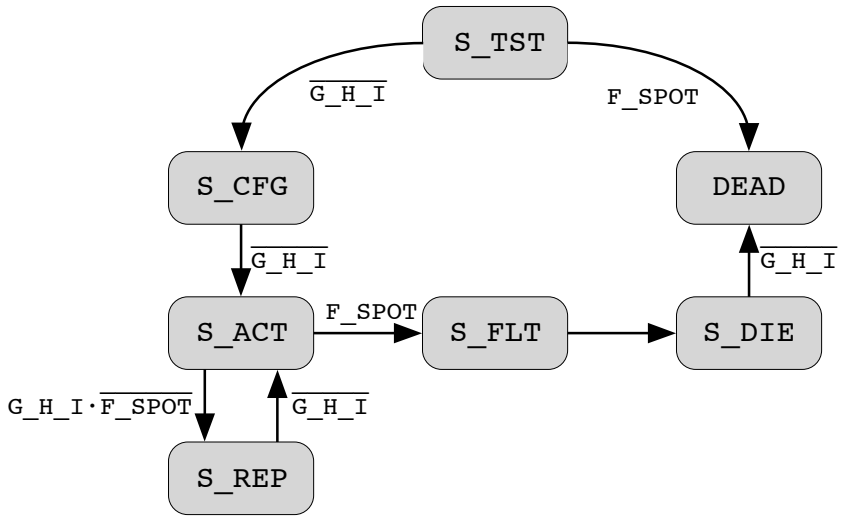

Figure B-8: The state graph for the operation of the state machine.

they pull the global hold line to 1 as long as their configuration register is not full. Once configured, they release the global hold line. Once the global hold line goes to 0 (i.e., when all the elements have been configured), the array begins operation and all elements go to state S_ACT.

- S_ACT (110) is the active state. All elements, once configured, remain in this state and execute their function until the circuit is reset or the repair mechanism is activated.
- DEAD (111) is the state of faulty elements after the reconfiguration is complete. Elements in this state are not part of the operation of the array, and the only exit from this state is through a global reset of the FPGA.

Among the other signals generated in this subcircuit, we will mention:

- G_K_O, a semi-global signal which propagates the KILL command (subsection 4.3.5) throughout a column of blocks;
- R_COL_O, which propagates downwards the information that the column contains spare elements;
- R_C_EO, which is set to 1 if the current element is or has been involved in the repair process;
- R_C_WO, which is set to 1 to signal the completion of the repair process;
- V_DEV and H_DEV, which control the rerouting of the connections;
- REG_EN, which enables the shifting of the array, either during the configuration phase or during the repair process;
- REG_RST, which resets the configuration register.

Apart from the above, the subcircuit also contains some intermediary control signals and some control logic for the 7-segment displays in the Biodule.

B.1.5 The Programmable Function

The subcircuit implementing the programmable function (Fig. B-9) is a straightforward implementation of the layout shown in Fig. 4-14. It consists of two copies of the original function plus some test logic.

The left (Fig. B-10) and the right (Fig. B-11) copies are identical, except for the different positioning of the faults. The layout corresponds exactly to the one shown in Fig. 4-2, with the exception of a slightly modified input selector, which nevertheless performs the same function. Note the relatively complex control logic for the flip flop, required to allow it to be set to a default value stored in the configuration (`FF_DEF`) and to be chained with the configuration register (`FF_CFG`).

As far as the test logic is concerned (Fig. B-12), it is a direct implementation of the circuit described in subsection 4.2.3, with the comparator for fault detection, the third copy of the flip-flop, and the 2-out-of-3 majority function `FF_OUT`.

B.1.6 The Switch Block

There is not much to say about the switch block (Fig. B-13), which is a straightforward implementation of the circuit described in subsection 4.1.2. It allows any one of the four output busses to carry the signal coming from the three other input busses or the output `N_OUT` of the element.

B.1.7 The Configuration Register

The hardware implementation of our configuration register led us to one relatively important modification to the propagation mechanism for the bitstream. This modification (which in no way alters our overall approach) concerns the mechanism which fills the registers with the appropriate value.

In the mechanism we described in chapter 4, the registers within a block were chained together to form one long shift chain. This approach, while perfectly feasible, requires however a dedicated global line to signal the end of the block's configuration, as well as some relatively important amount of additional logic to detect such an event.

In order to avoid this penalty, we altered our mechanism so as to fill the registers in order (Fig. B-14): rather than filling all registers in series, we fill the first, then the second, and so on until the last in the block. The new mechanism does away with the need to detect and propagate the end of the configuration at the expense of an additional bit in the configuration register. In fact, with this approach we require a way to detect that a register has been completely filled, so as to proceed to the next. We found that the least expensive way to obtain such behavior was to set the first (head) bit of the configuration of each element to `1`: the end of the configuration is then signaled by the arrival of the `1` to the head of the register.

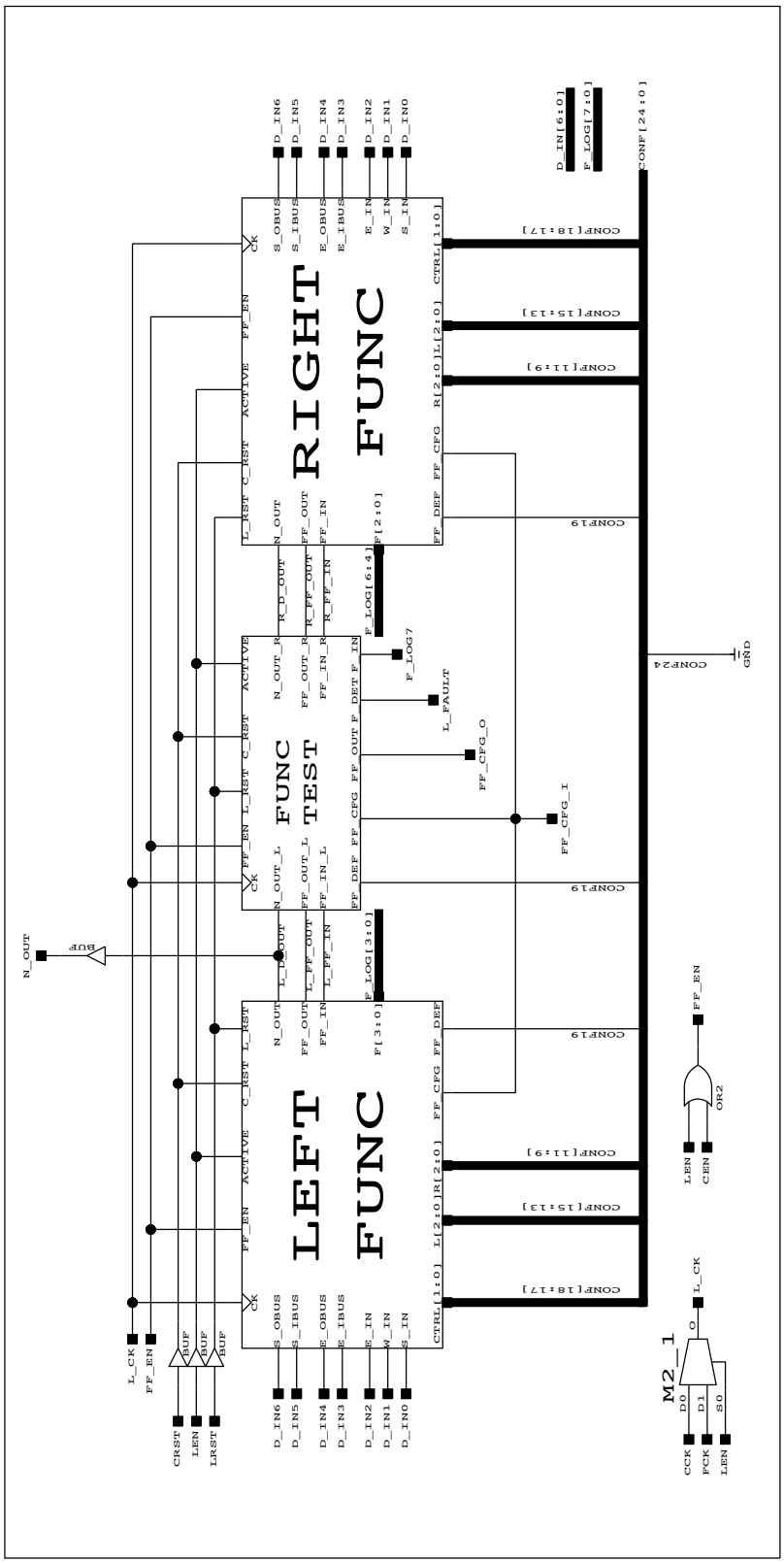

Figure B-9: The functional part of the element, including the test logic.

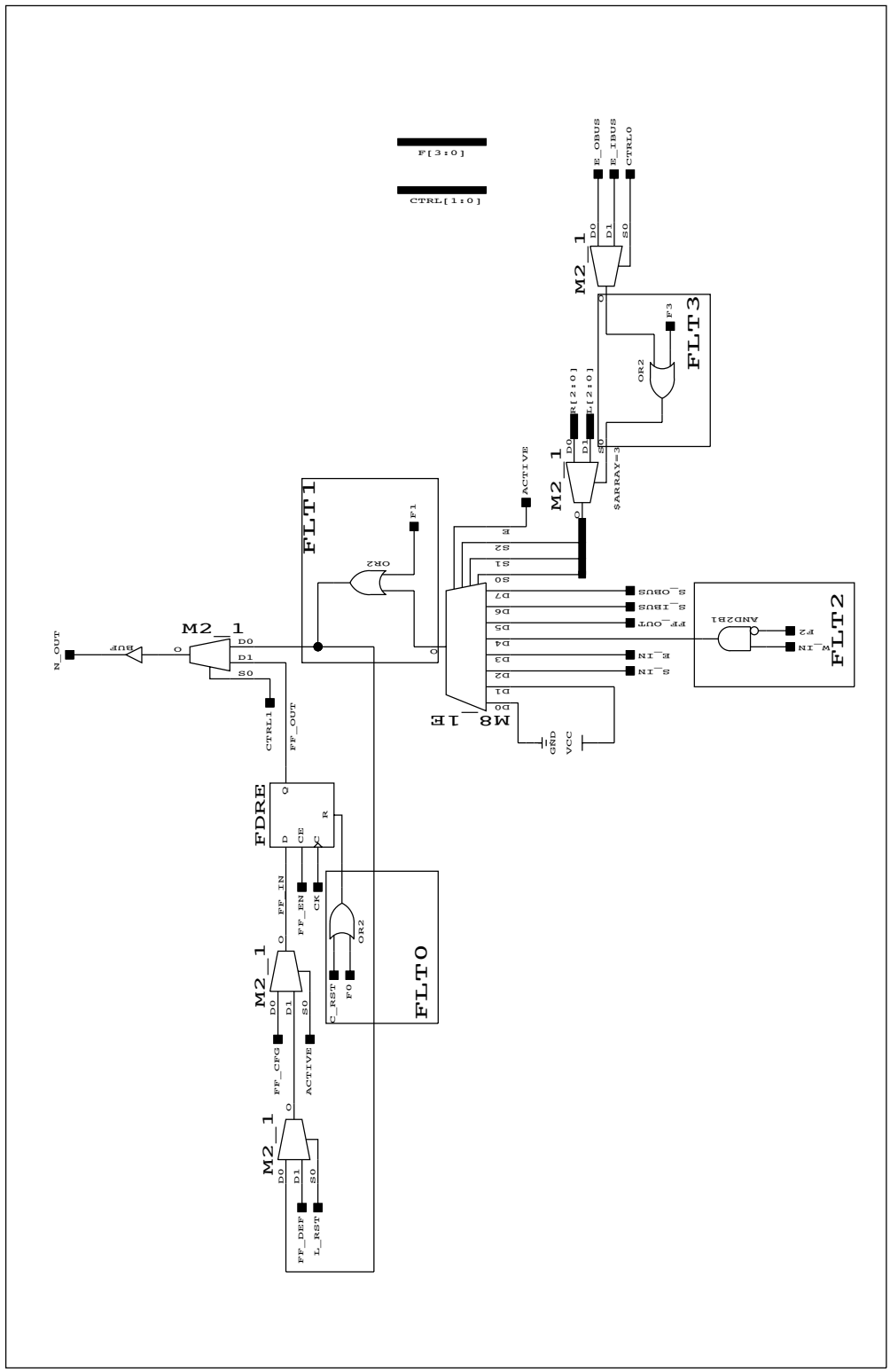

Figure B-10: The left copy of the MuxTree functional logic.

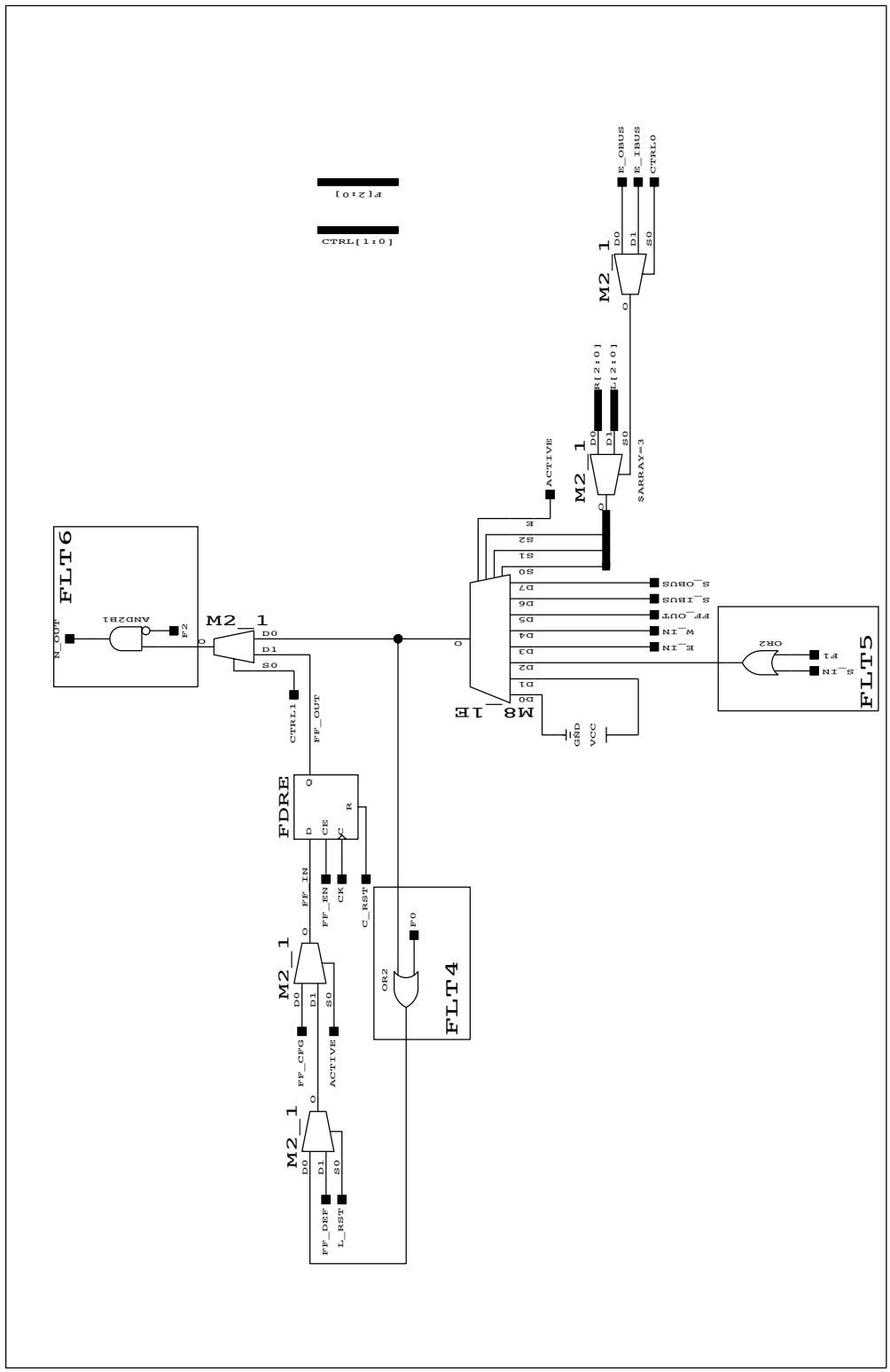

Figure B-11: The right copy of the MuxTree functional logic.

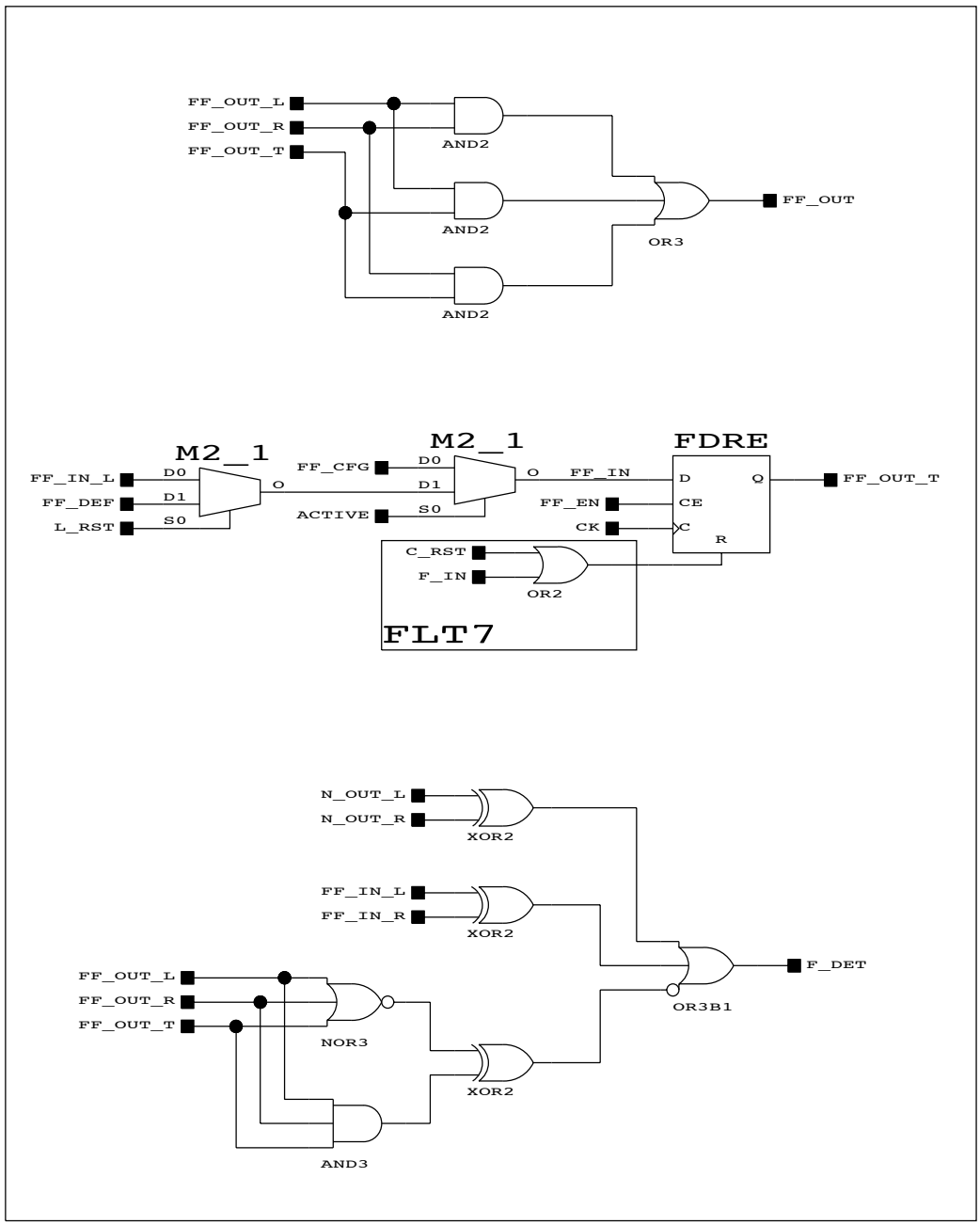

Figure B-12: The functional test logic.

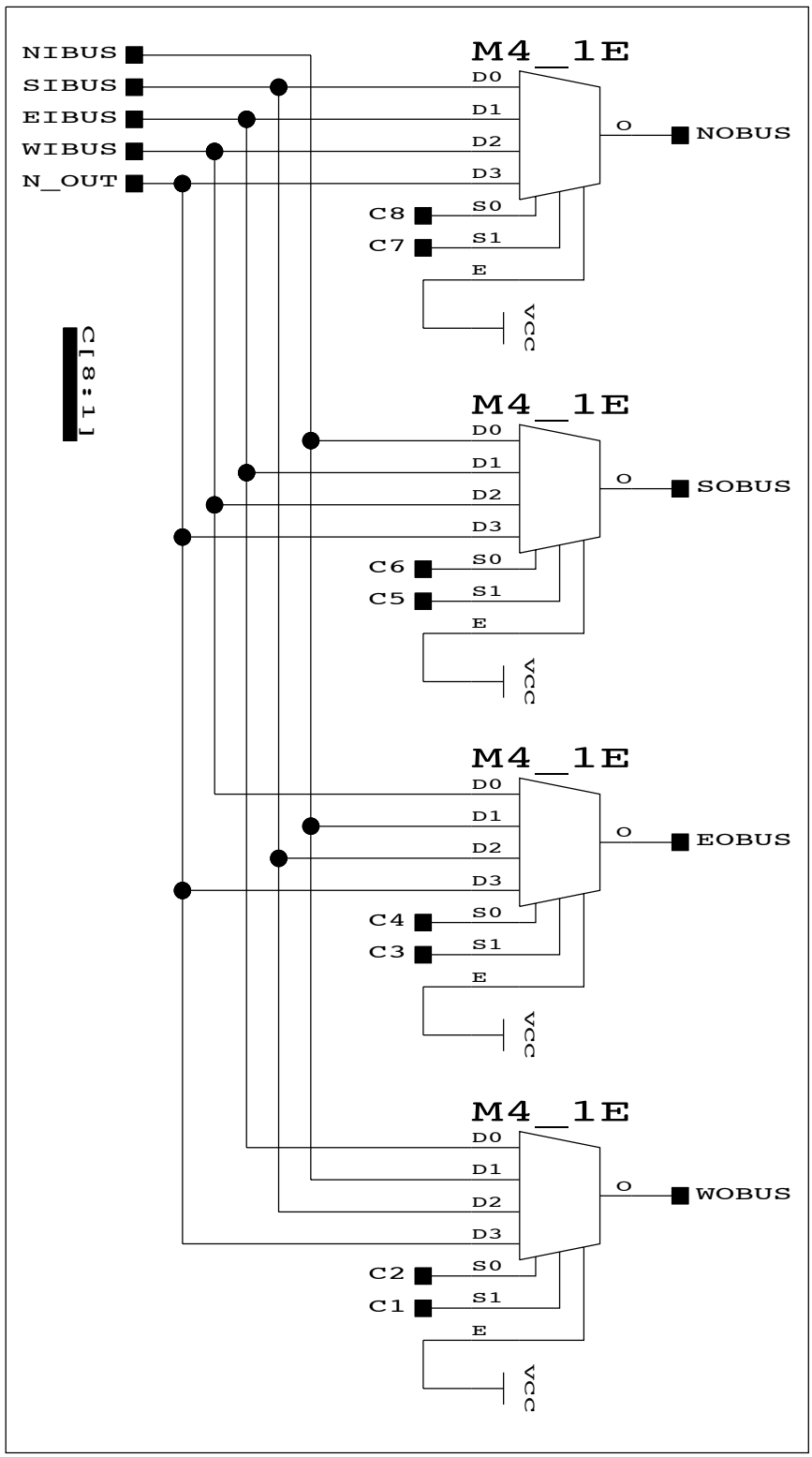

Figure B-13: The switch block for long-distance connections.

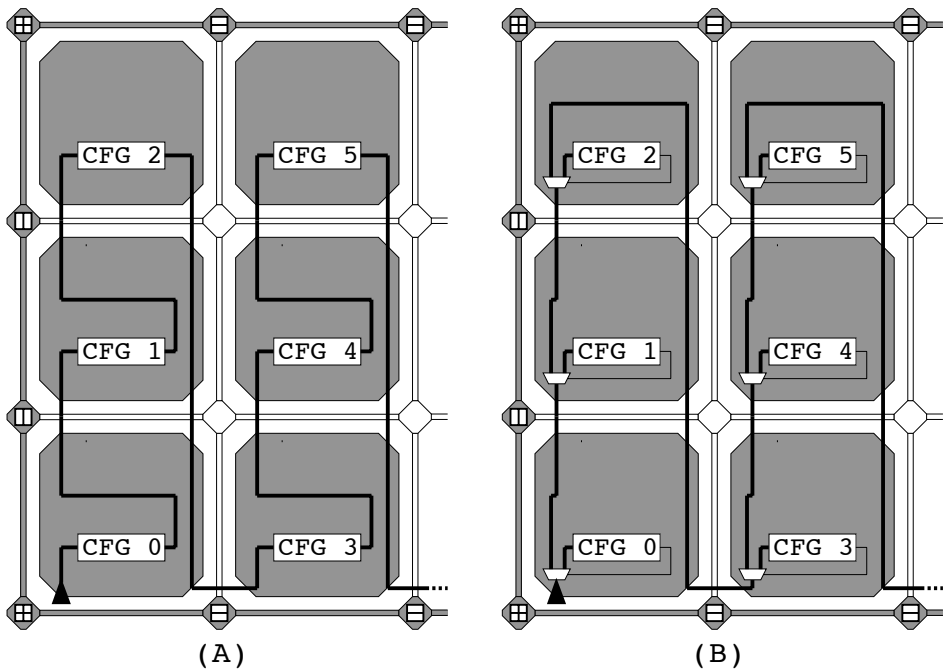

Figure B-14: The original (A) and the altered (B) mechanisms for propagating the configuration in each block.

Aside from this modification, the implementation of our configuration register (Fig. B-15) is identical to the design introduced in chapter 4. The register, which in this implementation is 24-bit wide[1], receives its configuration from one of three possible sources:

- if the array is expecting the test sequence of subsection 4.2.5 to arrive, all the registers will receive it in parallel from the global configuration line;
- if the element is on the bottom row of a block, or if the element is undergoing reconfiguration, the input will come from the west input line[2];
- if none of the above conditions is true, than the register will receive its configuration from the south input line.

Once the register is full (`C0=1`), the propagation path is fixed according to the patterns of Fig. 4-5:

- the output line to the north propagates the input line from the south unless the element is in the bottom row of the block, in which case it propagates the input from the west, or an entry point, in which case it propagates the global configuration line;

1. The extra bits are unused in the current implementation, but were introduced in view of possible future alterations in the design.
2. Note that the special case of the entry points was taken care of at the self-repair level (subsection B.1.3).

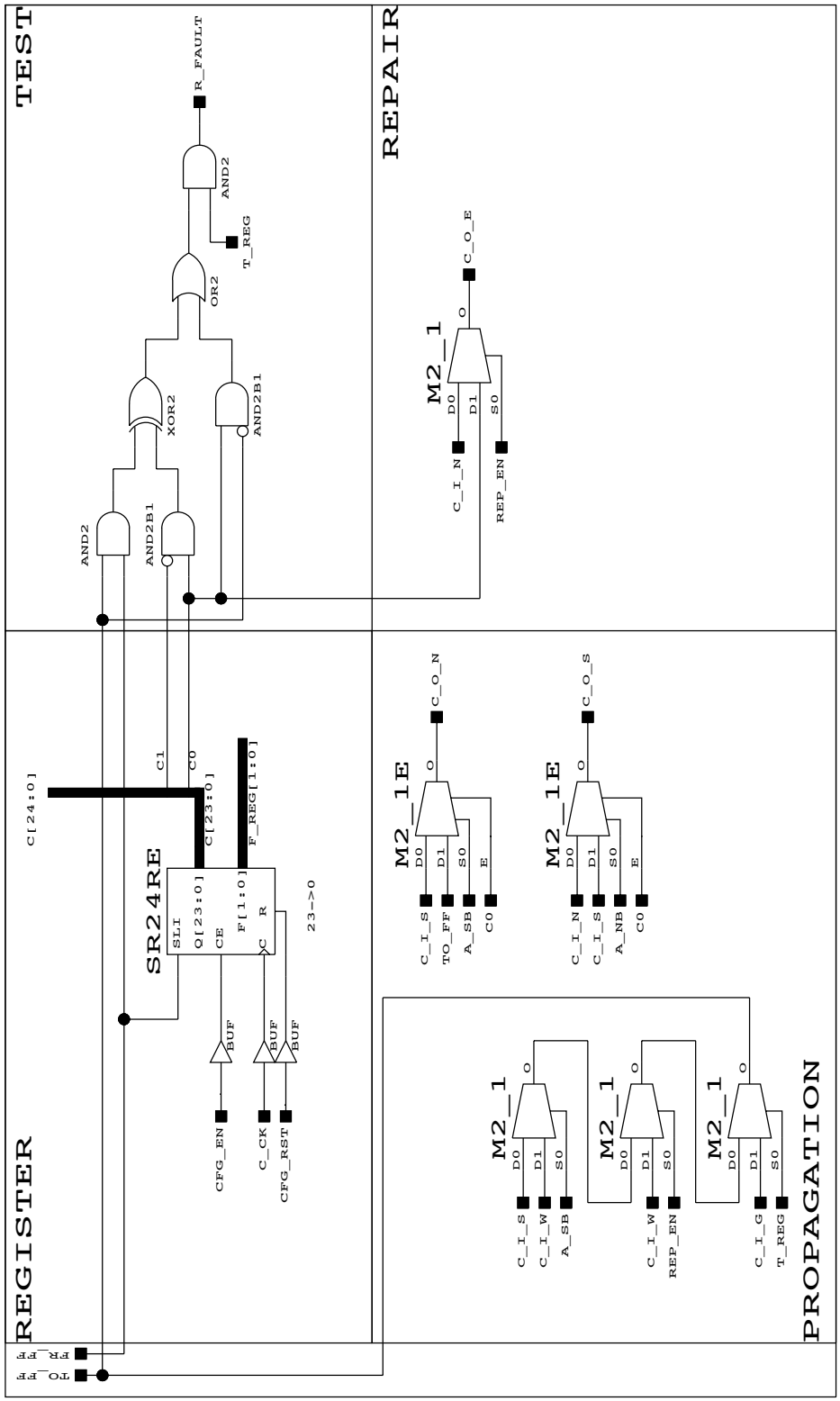

Figure B-15: The configuration register, including the propagation routing logic.

- the output line to the south propagates the input line from the north unless the element is in the top row of the block, in which case it propagates the input from the south input line;
- during configuration, the output line to the east propagates the input line from the north in all cases, while during self-repair it is used to shift the configuration of an element to its east neighbor.

Finally, the register subcircuit of Fig. B-15 contains the test logic described in subsection 4.2.5, slightly augmented to handle some Xilinx-related issues.

B.2 MuxNet

As we have seen, the Biodule 603 is the main hardware prototype we realized in order to test the validity of our mechanisms. However, the size of the circuit required to implement a single MuxTreeSR element prevents us from realizing systems which require more than a few logic elements. In the long term, we hope to overcome this difficulty through the realization of a dedicated VLSI circuit which will contain an important number of elements. In the short term, however, such a solution is not available to us.

To obtain a larger number of programmable elements, we investigated the possibility of exploiting a system based on an array of Xilinx FPGAs mounted on a single printed circuit board and configured so as to implement an array of MuxTreeSR elements. Such a system, while far from allowing the same density as a VLSI chip, would nevertheless allow us to obtain a much larger number of elements than an array of Biodules, particularly if we were not limited to a single MuxTreeSR element for each Xilinx chip.

The first step in the design of this system was therefore an analysis of the number of MuxTreeSR elements we can fit into a single Xilinx FPGA. To this end, we defined a layout consisting of a 4x4 array of our logic elements (Fig. B-16). Without attempting major optimizations in the layout of the elements[3], we removed the logic dedicated exclusively to the Biodule displays, and tried to determine the smallest Xilinx FPGA capable of containing the whole array. Running our design through the Xilinx routing software, we determined that the smallest FPGA which can hold the entire array is a XC4025HQ240. A system based on an array of such chips could thus allow us to obtain a fairly large array of MuxTreeSR elements, and indeed would be an interesting intermediate step in the creation of our VLSI circuit, likely to be realized as soon as we arrive at a quasi-definitive version of our FPGA.

3. The optimal solution to this kind of design would be to create a “hard macro” of a MuxTreeSR element, that is, define by hand the configuration of the circuit. Such a solution, however, would be extremely time-consuming.

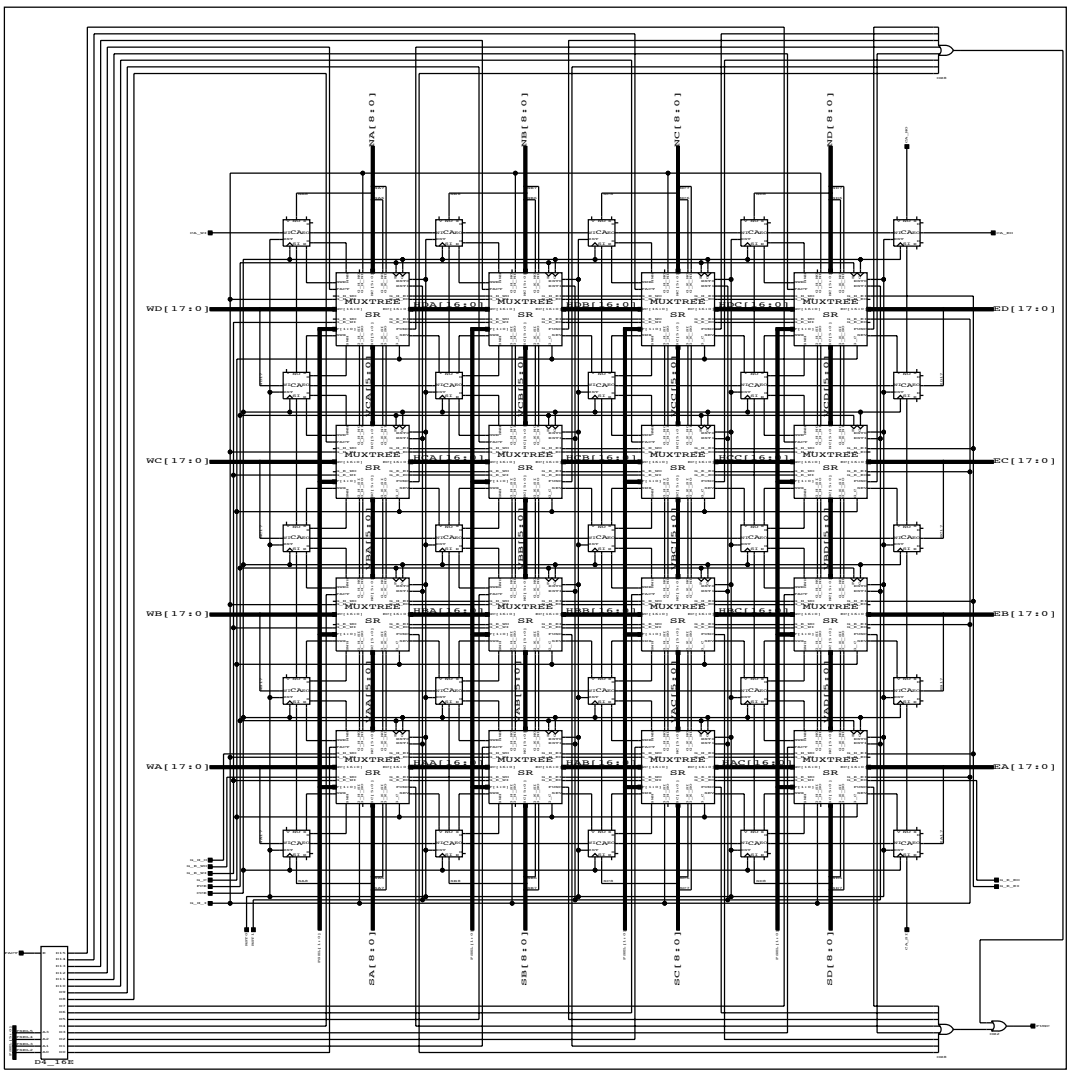

Figure B-16: Logic layout of a 4x4 array of MuxTreeSR elements.

REFERENCES

[1] M. Abramovici, M. A. Breuer, A. D. Friedman. *Digital Systems Testing and Testable Design*. Computer Science Press, New York, 1990.

[2] M. Abramovici, C. Stroud. "No-overhead BIST for FPGAs". In *Proc. 1st IEEE International On-Line Testing Workshop*, 1995, pp. 90-92.

[3] Actel Corporation. *ACT1 Series Data Book*. April 1996.

[4] S. B. Akers. "Binary Decision Diagrams". *IEEE Transactions on Computers*, c-27(6), June 1978, pp. 509-516.

[5] G.S. Almasi, A. Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, Redwood City, CA, 1989.

[6] Altera Corporation. *1998 Data Book*.

[7] Altera Corporation. "Means and Apparatus to Minimize the Effects of Silicon Processing Defects in Programmable Logic Devices". U.S. Patent 5592102.

[8] Altera Corporation. "Programmable Logic Devices with Spare Circuits for Replacement of Defects". U.S. Patent 5434514.

[9] M.A. Arbib, ed. *Handbook of Brain Theory and Neural Networks*. MIT Press, 1995.

[10] W. Asprey. *John von Neumann and the Origins of Modern Computing*. The MIT Press, Cambridge, MA, 1992.

[11] E.R. Banks. "Universality in Cellular Automata". In *Proc. IEEE 11th Annual Symposium on Switching and Automata Theory*, Santa Monica, CA, October 1970, pp. 194-215.

[12] J.-L. Beuchat, J.-O. Haenni. "Von Neumann's 29-State Cellular Automaton: A Hardware Implementation". *IEEE Trans. on Education*. Submitted.

[13] A. Boubekeur, J.-L. Patry, G. Saucier, J. Trilhe. "Configuring a Wafer-Scale Two-Dimensional Array of Single-Bit Processors". *IEEE Computer*, April 1992, pp. 29-39.

[14] S.D. Brown, R.J. Francis, J. Rose, Z.G. Vranesic. *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, Boston, 1992.

[15] E. Bruchez, J.-O. Haenni, E. Mosanya, E. Sanchez. "RENCO: un ordinateur de réseau reconfigurable". *Actes du XIIe Congrés "De Nouvelles Architectures pour les Communications"*, Paris, Dec. 1997, pp. 35-39.

[16] R.E. Bryant. "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams". *ACM Computing Surveys*, 24(3), 1992, pp. 293-318.

[17] A. Burks, ed. *Essays on Cellular Automata*. University of Illinois Press, Urbana, IL, 1970.

[18] J. Byl. “Self-Reproduction in Small Cellular Automata”. *Physica 34D*, pp.295-299, 1989.

[19] M. Chen, J.A.B. Fortes. “A Taxonomy of Reconfiguration Techniques for Fault-Tolerant Processor Arrays”. *IEEE Computer*, Jan. 1990, pp. 55-69.

[20] E.F. Codd. *Cellular Automata*. Academic Press, New York, 1968.

[21] A. Danchin. “Stabilisation fonctionnelle et épigénèse: une approche biologique de la genèse de l'identité individuelle”. In *L'identité*, Grasset, 1977, pp. 185-221.

[22] A. Danchin. “A Selective Theory for the Epigenetic Specification of the Monospecific Antibody Production in Single Cell Lines”. In *Ann. Immunol.*, Institut Pasteur, 1976, vol. 127C, pp. 787-804.

[23] N.J. Davis IV, F. Gail Gray, J.A. Wegner, S.E. Lawson, V. Murthy, T.S. White. “Reconfiguring Fault-Tolerant Two-Dimensional Array Architectures”. *IEEE Micro*, April 1994, pp. 60-68.

[24] S. Durand, C. Piguet. “FPGA with Selfrepair Capabilities”. In *Proc. ACM 2nd Int. Workshop on Field-Programmable Gate Arrays*, Feb. 1994, pp. 1-6.

[25] J.M. Emmert, D. Bhatia. “Partial Reconfiguration of FPGA Mapped Designs with Applications to Fault Tolerance and Yield Enhancement”. In *Proc. 7th Int. Workshop on Field-Programmable Logic and Applications*, London, Sept. 1997, pp. 141-150.

[26] D. Floreano. “Reducing Human Design and Increasing Adaptivity in Evolutionary Robotics”. In T. Gomi, ed., *Evolutionary Robotics*, AAI Books, Ontario, Canada, 1997, pp. 187-220.

[27] D. Floreano, J. Urzelai. “Evolution and Learning in Autonomous Robotic Agents”. In D. Mange, M. Tomassini, eds., *Bio-inspired Computing Machines: Towards Novel Computational Architectures,* Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998, pp. 317-346.

[28] D. B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press, Piscataway, NJ, 1995.

[29] R. A. Freitas, Jr., W. P. Gilbreath, eds. “Advanced Automation for Space Missions”. In *Proceedings of the 1980 NASA/ASEE Summer Study*, Scientific and Technical Information Branch (available from U.S. G.P.O.), Washington, D.C., 1980.

[30] B. Fritzke. “Growing Cell Structures - a Self-Organizing Network in k Dimensions”. *Artificial Neural Networks 2*, 1992.

[31] J. Gabay. “ASIC Technologies Boost Chip Density and Speed”. High Performance Systems, May 1990, pp. 32-53.

[32] M. Gardner. “The Fantastic Combinations of John Conway's New Solitaire Game ‘Life’”. *Scientific American*, vol. 223, no. 4, October 1970, pp. 120-123.

[33] M. Goeke, M. Sipper, D. Mange, A. Stauffer, E. Sanchez, M. Tomassini. “Online Autonomous Evolware”. In T. Higuchi, M. Iwata, W. Liu, eds., *Proc. 1st Int. Conference on Evolvable Systems: From Biology to Hardware*

(ICES96), Lecture Notes in Computer Science, vol. 1259, Springer-Verlag, Berlin, 1997, pp. 96-106.

[34] J.-O. Haenni, J.-L. Beuchat, E. Sanchez. "RENCO: A Reconfigurable Network Computer". In *Proc. FCCM '98*, Napa, CA, April 1998.

[35] F. Hanchek, S. Dutt. "Methodologies for Tolerating Cell and Interconnect Faults in FPGAs". *IEEE Transactions on Computers*, v. 47, n. 1, Jan. 1998.

[36] M. H. Hassoun. *Fundamentals of Artificial Neural Networks*. The MIT Press, Cambridge, MA, 1995.

[37] T. Higuchi, M. Iwata, I. Kajitani, H. Iba, Y. Hirao, T. Furuya, B. Manderick. "Evolvable Hardware and its Application to Pattern Recognition and Fault-Tolerant Systems". In E. Sanchez, M. Tomassini, eds., *Towards Evolvable Hardware,* Lecture Notes in Computer Science, Springer, Berlin, 1996, pp. 118-135.

[38] T. Hikage, H. Hemmi, K. Shimohara. "Hardware Evolution System Introducing Dominant and Recessive Heredity". In T. Higuchi, M. Iwata, W. Liu, eds., *Proc. 1st Int. Conference on Evolvable Systems: From Biology to Hardware (ICES96),* Lecture Notes in Computer Science, vol. 1259, Springer-Verlag, Berlin, 1997, pp. 423-436.

[39] J. E. Hopcroft, J. D. Ullman. *Introduction to Automata Theory Languages and Computation*. Addison-Wesley, Redwood City, CA, 1979.

[40] N.J. Howard, A.M. Tyrrell, N.M. Allinson. "The Yield Enhancement of Field-Programmable Gate Arrays". *IEEE Trans. on VLSI*, vol. 2, no. 1, March 1994, pp. 115-123.

[41] W.K. Huang, F. Lombardi. "An Approach for Testing Programmable/Configurable Field Programmable Gate Arrays". *IEEE VLSI Test Symposium*, 1996.

[42] C. Iseli. *Spyder: A Reconfigurable Processor Development System*. PhD. Thesis No. 1476, Computer Science Department, Swiss Federal Institute of Technology, Lausanne, 1996.

[43] C. Iseli, E. Sanchez. "Spyder: A SURE (SUperscalar and REconfigurable) Processor". *The Journal of Supercomputing*, 9, 1995, pp. 231-252.

[44] P. Jackson. *Introduction to Expert Systems*. Addison-Wesley, Redwood City, CA, 2nd edition, June 1990.

[45] N.K. Jha, S. Kundu. *Testing and Reliable Design of CMOS Circuits*. Kluwer Academic Publishers. Boston, MA, 1990.

[46] J. O. Kephart, G. B. Sorkin, David M. Chess, Steve R. White. "Fighting Computer Viruses". *Scientific American*, November 1997, vol. 277, No. 5, pp. 56-61.

[47] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill, New York, 2nd edition, 1978.

[48] J. R. Koza. *Genetic Programming*. The MIT Press, Cambridge, MA, 1992.

[49] J. R. Koza. *Genetic Programming II*. The MIT Press, Cambridge, MA,1994.

[50] J. R. Koza, F. H. Bennett III, D. Andre, M. A. Keane. "Automated {WYWIWYG} Design of Both the Topology and Component Values of Electrical

Circuits Using Genetic Programming". In *Genetic Programming 1996: Proceedings of the First Annual Conference*, The MIT Press, Cambridge, MA, 1996, pp. 123-131.

[51] J. Lach, W.H. Mangione-Smith, M. Potkonjak. "Efficiently Supporting Fault-Tolerance in FPGAs". *Proc. FPGA '98*, Monterey, CA, Feb. 1998, pp. 105-115.

[52] P.K. Lala. *Digital Circuit Testing and Testability.* Academic Press, San Diego, 1997.

[53] C. G. Langton. "Self-Reproduction in Cellular Automata". *Physica 10D*, 1984, pp. 135-144.

[54] C. Lee. "Synthesis of a Cellular Computer". In *Applied Automata Theory*, Academic Press, London, 1968, pp 217-234.

[55] H.R. Lewis, C.H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[56] F. Lombardi, M.G. Sami, R. Stefanelli. "Reconfiguration of VLSI Arrays by Covering". *IEEE Transactions on CAD*, vol. 8, no. 9, Sept. 1989, pp. 952-965.

[57] D. Mange. *Microprogrammed Systems: An Introduction to Firmware Theory.* Chapman & Hall, London, 1992. (First published as *Systémes Microprogrammés: une introduction au magiciel*, Presses Polytechniques et Universitaires Romandes, 1990).

[58] D. Mange, D. Madon, A. Stauffer, G. Tempesti. "Von Neumann Revisited: A Turing Machine with Self-Repair and Self-Reproduction Properties". *Robotics and Autonomous Systems*, Vol. 22, No. 1, 1997, pp. 35-58.

[59] D. Mange, M. Goeke, D. Madon, A. Stauffer, G. Tempesti, S. Durand. "Embryonics: A New Family of Coarse-Grained Field-Programmable Gate Array with Self-Repair and Self-Reproducing Properties". In E. Sanchez, M. Tomassini, eds., *Towards Evolvable Hardware,* Lecture Notes in Computer Science, Springer, Berlin, 1996, pp. 197-220.

[60] D. Mange, M. Tomassini, eds. *Bio-inspired Computing Machines: Towards Novel Computational Architectures*. Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998.

[61] P. Marchal, A. Stauffer. "Binary Decision Diagram Oriented FPGAs". In: *Proc. FPGA '94, 2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, Berkeley, CA, February 1994, pp. 1-10.

[62] P. Marchal, P. Nussbaum, C. Piguet, S. Durand, D. Mange, E. Sanchez, A. Stauffer, G. Tempesti. "Embryonics: The Birth of Synthetic Life". In E. Sanchez, M. Tomassini, eds., *Towards Evolvable Hardware*, Lecture Notes in Computer Science, Springer, Berlin, 1996, pp. 166-197.

[63] E. J. McCluskey. *Logic Design Principles with Emphasis on Testable Semicustom Circuits*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[64] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, Berlin, 3rd ed., 1996.

[65] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, 1996.

[66] T. Miyamori, K. Olukotun. "A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications". In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM98),* Napa, CA, April 1998.

[67] J.M. Moreno, *VLSI Architectures for Evolutive Neural Models*, PhD. Thesis, Universitat Politecnica de Catalunya, Barcelona, 1994.

[68] E. Mosanya, J.-M. Puiatti, E. Sanchez. "Hardware Implementation of Generalized Profile Search on the GENSTORM Machine". In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM98),* Napa, CA, April 1998.

[69] R. Negrini, M.G. Sami, R. Stefanelli. *Fault Tolerance Through Reconfiguration in VLSI and WSI Arrays*. The MIT Press, Cambridge, MA, 1989.

[70] R. Negrini, M.G. Sami, R. Stefanelli. "Fault Tolerance Techniques for Array Structures Used in Supercomputing". *IEEE Computer*, vol. 19, no. 2, February 1986, pp. 78-87.

[71] F. Nourai, R.S. Kashef. "A Universal Four-State Cellular Computer". In *IEEE Transactions on Computers*, c-24(8), August 1975, pp. 766-776.

[72] A. Ohta, T. Isshiki, H. Kunieda. "New FPGA Architecture for Bit-Serial Pipeline Datapath". In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM98)*Napa, CA, April 1998.

[73] M. Peercy, P. Banerjee. "Fault Tolerant VLSI Systems". In: *Proceedings of the IEEE*, Vol. 81, No. 5, May 1993, pp. 745-758.

[74] A. Perez-Uribe. "Artificial Neural Networks: Algorithms and Hardware Implementation". In D. Mange, M. Tomassini, eds., *Bio-inspired Computing Machines: Towards Novel Computational Architectures,* Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998, pp. 289-316.

[75] A. Perez-Uribe, E. Sanchez. "The FAST Architecture: A Neural Network with Flexible Adaptable-Size Topology". In *Proc. 5th International Conference on Microelectronics for Neural Networks and Fuzzy Systems*, Lausanne, Switzerland, February 1996.

[76] A. Perez-Uribe, E. Sanchez. "FPGA Implementation of an Adaptable-Size Neural Network". In *Proc. International Conference on Artificial Neural Networks ICANN96*, Bochum, Germany, July 1996.

[77] J.-Y. Perrier, M. Sipper, J. Zahnd. "Toward a Viable, Self-Reproducing Universal Computer". *Physica 97D*, pp. 335-352, 1996.

[78] U. Pesavento. "An Implementation of von Neumann's Self-Reproducing Machine". *Artificial Life*, 2(4), 1995, pp. 337-354.

[79] C. Piguet, P. Weiss, P. Marchal. "Reliability and Self-Repair of Integrated Circuits". In *Proc. PATMOS '95*, Paper S8.2, Oldenburg, Germany, Oct. 1995, pp. 316-328.

[80] R. Rajsuman. *Digital Hardware Testing: Transistor-Level Fault Modeling and Testing*. Artech House, Boston, MA, 1992.

[81] T. S. Ray. "An Approach to the Synthesis of Life". In *Artificial Life II*, Addison-Wesley, Redwood City, CA, 1992, pp. 371-408.

[82] J.A. Reggia, S.A. Armentrout, H.-H. Chou, Y. Peng. "Simple Systems That Exhibit Self-Directed Replication". *Science*, Vol.259, 26 February 1993, pp. 1282-1287.

[83] E. Sanchez, M. Tomassini, eds. *Towards Evolvable Hardware*. Lecture Notes in Computer Science, Springer, Berlin, 1996.

[84] E. Sanchez, D. Mange, M. Sipper, M. Tomassini, A. Perez-Uribe, A. Stauffer. "Phylogeny, Ontogeny, and Epigenesis: Three Sources of Biological Inspiration for Softening Hardware". In T. Higuchi, M. Iwata, W. Liu, eds., *Proc. 1st Int. Conference on Evolvable Systems: From Biology to Hardware (ICES96),* Lecture Notes in Computer Science, vol. 1259, Springer-Verlag, Berlin, 1997, pp. 35-54.

[85] A. Shibayama, H. Igura, M. Mizuno, M. Yamashina. "An Autonomous Reconfigurable Cell Array for Fault-Tolerant LSIs". In: *Proc. 1997 IEEE International Solid-State Circuits Conference*, February 1997.

[86] M. Sipper. "Co-Evolving Non-Uniform Cellular Automata to Perform Computations". *Physica D* 92 (1996), pp. 193-208.

[87] M. Sipper. *Evolution of Parallel Cellular Machines: The Cellular Programming Approach*. Springer-Verlag, Berlin, 1997.

[88] M. Sipper. "If the Milieu Is Reasonable: Lessons from Nature on Creating Life". *Journal of Transfigural Mathematics*, 3(1):7-22, 1997.

[89] M. Sipper, D. Mange, A. Stauffer. "Ontogenetic Hardware". *BioSystems* 44 (1997), pp. 193-207.

[90] M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Perez-Uribe, A. Stauffer. "An Introduction to Bio-Inspired Machines". In D. Mange, M. Tomassini, eds., *Bio-inspired Computing Machines: Towards Novel Computational Architectures,* Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998, pp. 1-12.

[91] E. H. Spafford. "Computer Viruses - A Form of Artificial Life?". In C.G. Langton, C. Taylor, J. D. Farmer, S. Rasmussen, eds., *Artificial Life II*, Addison-Wesley, Redwood City, CA, 1992, pp. 727-745.

[92] A. Stauffer. "Membrane Building and Binary Decision Machine Implementation". Technical Report 247, Computer Science Department, EPFL, Lausanne, 1997.

[93] A. Stauffer, D. Mange, M. Goeke, D. Madon, G. Tempesti, S. Durand, P. Marchal, C. Piguet. "MICROTREE: Towards a Binary Decision Machine-Based FPGA with Biological-like Properties". In *Proc. IFIP International Workshop on Logic and Architecture Synthesis*, Grenoble, December 1996, pp. 103-112.

[94] A. Stauffer, D. Mange, E. Sanchez, G. Tempesti, S. Durand, P. Marchal, C. Piguet. "Embryonics: Towards New Design Methodologies for Circuits with

Biological-like Properties". In *Proc. International Workshop on Logic and Architecture Synthesis*, Grenoble, December 1995, pp. 299-306.

[95] C. Stroud, S. Konala, M. Abramovici. "Using ILA Testing for BIST in FPGAs". *Proc. 2nd IEEE International On-Line Testing Workshop*, Biarritz, July 1996.

[96] C. Stroud, S. Konala, P. Chen, M. Abramovici. "Built-In Self-Test of Logic Blocks in FPGAs". In *Proc. IEEE 14th VLSI Test Symposium*, 1996, pp. 387-392.

[97] G. Tempesti. "A New Self-Reproducing Cellular Automaton Capable of Construction and Computation". *Proc. 3rd European Conference on Artificial Life*, Lecture Notes in Artificial Intelligence, 929, Springer Verlag, Berlin, 1995, pp. 555-563.

[98] G. Tempesti, D. Mange, A. Stauffer. "A Robust Multiplexer-Based FPGA Inspired by Biological Systems". *Journal of Systems Architecture: Special Issue on Dependable Parallel Computer Systems*, 43(10), 1997.

[99] A. Tettamanzi, M. Tomassini."Evolutionary Algorithms and their Applications". In D. Mange, M. Tomassini, eds., *Bio-inspired Computing Machines: Towards Novel Computational Architectures,* Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998, pp. 59-98.

[100] A. Thompson. "Silicon Evolution". In *Genetic Programming 1996: Proceedings of the First Annual Conference*, The MIT Press, Cambridge, MA, 1996, pp. 444-452.

[101] T. Toffoli and N. Margolus. *Cellular Automata Machines*. The MIT Press, Cambridge, MA, 1987.

[102] N. Tsuda, T. Satoh. "Hierarchical Redundancy for a Linear-Array Switching Chip". In *Proc. 2nd IFIP Workshop on WSI*, Brunel, Sept. 1987.

[103] S. Trimberger, ed. *Field-Programmable Gate Array Technology.* Kluwer Academic Publishers, Boston, 1994.

[104] J. von Neumann. *The Theory of Self-Reproducing Automata*. A. W. Burks, ed. University of Illinois Press, Urbana, IL, 1966.

[105] M. Wang, M. Cutler, S.Y.H. Su. "On-Line Error Detection and Reconfiguration with Two-Level Redundancy". In *Proc. COMPEURO 87*, Hamburg, 1987, pp. 703-706.

[106] S.-J. Wang, T.-M. Tsai. "Test and Diagnosis of Faulty Logic Blocks in FPGAs". *Proc. ICCAD 97*, pp. 722-727.

[107] *Webster's New Universal Unabridged Dictionary*, 2nd Ed., Simon & Schuster, 1983.

[108] P.H. Winston. *Artificial Intelligence*. Addison-Wesley, Reading, MA, 3rd edition, 1992.

[109] S. Wolfram. *Cellular Automata and Complexity.* Addison-Wesley, Reading, MA, 1994.

[110] *WordNet Online Lexical Database* (http://www.cogsci.princeton.edu/~wn).

[111] Xilinx Corporation. *XC4000E and XC4000X Series Data Book*. November 1997.

[112] Xilinx Corporation. *XC5200 Field Programmable Gate Arrays Data Book*. December 1997.

[113] Xilinx Corporation. *XC6200 Series Data Book*. April 1997.

CURRICULUM VITAE

Gianluca Tempesti

Address

Logic Systems Laboratory	Phone:	+41-21-693 2676 (W)
EPFL-DI-LSL, INN-Ecublens		+41-21-636 3303 (H)
CH-1015 Lausanne	Fax:	+41-21-693 3705
Switzerland	Email:	tempesti@di.epfl.ch
Citizenship: Italian	URL:	http://lslwww.epfl.ch

Work Experience

1994-Now	Swiss Federal Institute of Technology at Lausanne (EPFL) Lausanne, Switzerland Research and Teaching Assistant, Logic Systems Laboratory

Education

1995-Now	Swiss Federal Institute of Technology at Lausanne (EPFL) Lausanne, Switzerland Ph.D. in Computer Science and Engineering (May 1998)
1991-1993	University of Michigan at Ann Arbor Ann Arbor, Michigan, USA Masters Degree in Computer Science and Engineering
1987-1991	Princeton University Princeton, New Jersey, USA Bachelor of Science in Electrical (Computer) Engineering

Professional experience

Areas of Expertise:
- Design and development of FPGA circuits
- Logic systems design using FPGAs (Xilinx, Actel)
- Built-in self-test and self-repair techniques
- Bio-inspired electronic systems
- High-performance processor architectures
- Massively parallel computer systems
- Software simulation and test of logic circuits
- Some experience in the design of PCBs
- Development of complex cellular automata

Software Packages: ViewLogic, Xilinx and Actel tools, Synopsis, Exemplar.

Computer Languages: C, some C++ and assembly, VHDL, parallel C.

Languages: English, Italian -- excellent; French -- very good.

List of Publications

G. Tempesti, D. Mange, A. Stauffer. "Self-replicating Multicellular Automata". *Artificial Life Journal*. Submitted.

D. Mange, E. Sanchez, A. Stauffer, G. Tempesti. "Embryonics: A New Methodology for Designing Field-Programmable Gate Arrays with Self-Repair and Self-Reproducing Properties". *IEEE Transactions on VLSI Systems*. Accepted.

D. Mange, A. Stauffer, G. Tempesti. "The Embryonics Project: a Machine Made of Artificial Cells". *Rivista di Biologia-Biology Forum*. Accepted.

G. Tempesti. *A Self-Repairing Multiplexer-Based FPGA Inspired by Biological Processes*. Ph.D. Thesis, EPFL, Lausanne, 1998.

D. Mange, A. Stauffer, G. Tempesti. "Multiplexer-Based Cells". In D. Mange and M. Tomassini, eds., *Bio-Inspired Computing Machines*, Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998, pp. 121-165.

D. Mange, A. Stauffer, G. Tempesti. "Binary Decision Machine-Based Cells". In D. Mange and M. Tomassini, eds., *Bio-Inspired Computing Machines*, Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998, pp. 183-216.

D. Mange, A. Stauffer, G. Tempesti. "Self-Repairing Molecules and Cells". In D. Mange and M. Tomassini, eds., *Bio-Inspired Computing Machines*, Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998, pp. 217-267.

S. Durand, P. Marchal, P. Nussbaum, C. Piguet, D. Mange, E. Sanchez, A. Stauffer, G. Tempesti. "Life Organization as a Source of Inspiration for Self-Repairing VLSI". *Medical & Biological Engineering & Computing*, Vol. 35, Suppl. Part 1, 1997, p. 87.

D. Mange, D. Madon, A. Stauffer, G. Tempesti. "Von Neumann Revisited: A Turing Machine with Self-Repair and Self-Reproduction Properties". *Robotics and Autonomous Systems*, Vol. 22, No 1, 1997, pp. 35-58.

G. Tempesti, D. Mange, A. Stauffer. "A Robust Multiplexer-Based FPGA Inspired by Biological Systems". *Journal of Systems Architecture: Special Issue on Dependable Parallel Computer Systems*, EUROMICRO, 43(10), 1997.

G. Tempesti, D. Mange, A. Stauffer. "A Self-Repairing FPGA Inspired by Biology". In *Proc. 3rd IEEE International On-Line Testing Workshop*, pp.191-195. IEEE Computer Society, 1997.

A. Stauffer, D. Mange, M. Goeke, D. Madon, G. Tempesti, S. Durand, P. Marchal, P. Nussbaum. "FPPA: A Field-Programmable Processor Array with Biological-like Properties". In *Reconfigurable Architectures: High Performance by Configware*, ITpress-Verlag, Bruchsal, 1997, pp. 45-48.

P. Marchal, P. Nussbaum, C. Piguet, S. Durand, D. Mange, E. Sanchez, A. Stauffer, G. Tempesti. "Embryonics: The Birth of Synthetic Life". In E. Sanchez

and M. Tomassini, eds., *Towards Evolvable Hardware*, Springer-Verlag, Berlin, 1996.

D. Mange, M. Goeke, D. Madon, A. Stauffer, G. Tempesti, S. Durand. "Embryonics: A New Family of Coarse-Grained Field-Programmable Gate Array with Self-Repair and Self-Reproducing Properties". In E. Sanchez and M. Tomassini, eds., *Towards Evolvable Hardware*, Springer-Verlag, Berlin, 1996, pp.197-220.

P. Marchal, P. Nussbaum, C. Piguet, S. Durand, D. Mange, E. Sanchez, A. Stauffer, G. Tempesti. "Genomic Cellular Automata Transposed on Silicon: Experiments in Synthetic Life". In R. Cuthbertson, M. Holcombe, R. Paton, eds., *Computation in Cellular and Molecular Biological Systems*, World Scientific, Singapore, 1996.

D. Mange, M. Goeke, D. Madon, A. Stauffer, G. Tempesti, S. Durand, P. Marchal, P. Nussbaum. "Embryonics: A New Family of Coarse-Grained Field-Programmable Gate Array with Self-Repair and Self-Reproducing Properties". *Proc. ISCAS '96*, IEEE, Vol. 4, 1996, pp. 25-28.

A. Stauffer, D. Mange, M. Goeke, D. Madon, G. Tempesti, S. Durand, P. Marchal, C. Piguet. "MICROTREE: Towards a Binary Decision Machine-based FPGA with Biological-like Properties". *Proc. International Workshop on Logic and Architecture Synthesis*, Institut national polytechnique de Grenoble, December 16-18, 1996, pp. 103-112.

D. Mange, D. Madon, E. Sanchez, A. Stauffer, G. Tempesti, S. Durand, P. Marchal, C. Piguet. "BIOWATCH: une montre autoréparable et autoreproductrice". *Proc. 6ème Congrès Européen de Chronométrie*, Bienne, 17-18 Oct. 1996, pp. 39-42.

A. Stauffer, D. Mange, E. Sanchez, G. Tempesti, S. Durand, P. Marchal, C. Piguet. "Embryonics: Towards New Design Methodologies for Circuits with Biological-like Properties". *Proc. International Workshop on Logic and Architecture Synthesis*, Grenoble, December 18-19, 1995, pp. 299-306.

P. Marchal, P. Nussbaum, C. Piguet, S. Durand, D. Mange, E. Sanchez, A. Stauffer, G. Tempesti. "Genomic Cellular Automata Transposed in Silicon: Experiments in Synthetic Life". *Proc. International Workshop on Information Processing in Cells and Tissues (IPCAT '95)*, Liverpool, Sept. 6-8, 1995, pp. 1-14.

G. Tempesti. "A New Self-Reproducing Cellular Automaton Capable of Construction and Computation". *Advances in Artificial Life, Proc. 3rd European Conference on Artificial Life*, Granada, Spain, June 4-6, 1995, Lecture Notes in Artificial Intelligence, 929, Springer Verlag, Berlin, 1995, pp. 555-563.

D. Mange, S. Durand, E. Sanchez, A. Stauffer, G. Tempesti, P. Marchal, C. Piguet. "A New Paradigm for Developing Digital Systems Based on a Multi-Cellular Organization". *Proc. IEEE International Symposium on Circuits and Systems (ISCAS '95)*, Seattle, April 30-May 3, 1995, Vol. 3, pp. 2193-2196.