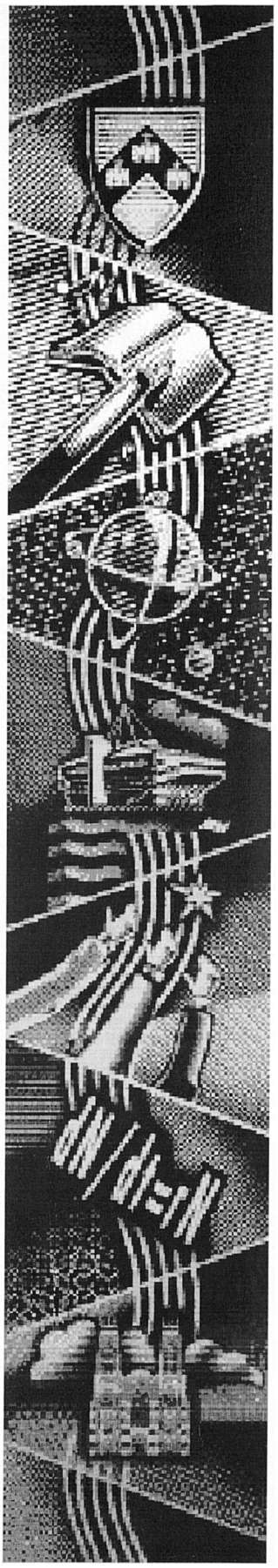


EMBRYONICS:
A BIO-INSPIRED FAULT-TOLERANT MULTICELLULAR SYSTEM

DPhil Thesis by

César A. Ortega-Sánchez

THE UNIVERSITY *of York*

Department of Electronics

Bio-Inspired and Bio-Medical Engineering Group

May, 2000

ABSTRACT

This thesis presents the research undertaken on a novel fault-tolerant cellular architecture with reconfiguration properties inspired by mechanisms found in natural multicellular systems. The architecture is called Embryonics (Embryology + Electronics). Embryonics proposes the application of mechanisms that take place during the embryological development of multicellular organisms to improve the reliability of 2-D silicon-based programmable cellular architectures. The basic embryonic cell performs a selection function. Logic functions are implemented in embryonic arrays by constructing networks of selectors that represent them. Three examples of the application of embryonic arrays are presented. To formally verify embryonic array's fault tolerance, mathematical reliability models for different embryonic reconfiguration strategies are derived. It is demonstrated that embryonic distributed systems possess, in the majority of cases, better reliability characteristics than equivalent centralised systems.

To you, Gigí

With all my love

ACKNOWLEDGEMENTS

It is impossible to express my gratitude individually to every person that contributed to both, the technical and the human aspects of my research. I will try to give credit to those without whom these three years in York would have been less fruitful.

First of all, to my wife and daughter. What would I be without you? You, who have always had a kiss to give and a nice word to say when I most needed them. Thank you for choosing to be my companions in this not always placid trip called life. I love you.

To my parents, who more than once were able to sacrifice their present to provide me with a future. I am proud to belong to you. Be sure that all those teachings in Tlatelolco and Marina Nacional have not been forgotten. I treasure them, and they will always guide me through life. God bless you.

To my brothers and sister, who have always believed in my potential and have always made my life happier. I thank you for your unconditional support.

To Andy Tyrrell, "The Boss", whose immaculate example and great sense of humour taught me that it is possible to be a respectable academic and a great human being at the same time. Thank you for your invaluable support and your always-opportune guidance.

To my young lab mates: Alex, Barry, Daryl, Demian, Gordon, Mic, Mike, Paul, Richard and Tony. A hundred more pages would be needed if I had to list every occasion they helped me out. Thanks for the laughs and the time you shared with this "old chap".

To my dear friends and colleagues Daniel Mange, Gianluca Tempesti, Eduardo Sanchez and Moshe Sipper from the Polytechnic of Lausanne, Switzerland. Thank you for the encouragement and unconditional support that you have always given to me. It has been a real privilege having a professional liaison with you. Merci beaucoup!

To my brother-in-law Macario Schettino for his opportune comments and help during the most arduous part of my research. Thank you Doc.

To David Howard and Steve Smith. They also contributed to make the dream come true.

To my friends from Angola, Argentina, Brazil, England, France, Germany, Greece, Holland, Iraq, Italy, Mexico, Portugal, Spain and Venezuela, who have been our family for the past three years. For your company, your solidarity and, of course, your food, you will always be remembered.

Last but not least, to the financial sponsors: The National Council for Science and Technology (CONACyT) and the Institute of Electrical Research through the Bank of Mexico. Thanks for believing in me. I am looking forward to demonstrate that paying for my professional fulfilment has been a good investment.

DECLARATION

This thesis is submitted for the degree of Doctor of Philosophy. All work presented is entirely my own work, except where explicitly referenced.

Part of this work has been presented in the following journals, conferences and colloquia:

1. Ortega C. and Tyrrell A., "Fault-tolerant Systems: The way Biology does it!", Proceedings Euromicro 97, Budapest, IEEE CS Press, September, 1997, pp.146-151
2. Ortega C. and Tyrrell A., " Biologically Inspired Reconfigurable Hardware for Dependable Applications", IEE Colloquium on Hardware Systems for Dependable Applications, London, November, 1997, Digest No: 97/335
3. Ortega C. and Tyrrell A., "Design of a Basic Cell to Construct Embryonic Arrays", IEE Procs. on Computers and Digital Techniques, Vol.145-3, May, 1998, pp.242-248
4. Ortega C. and Tyrrell A., "Evolvable Hardware for Fault-Tolerant Applications", IEE Colloquium on Evolvable Hardware Systems, London, March, 1998, Digest No: 98/233
5. Ortega C. and Tyrrell A., "Biologically Inspired Real-Time Reconfiguration Technique for Processor Arrays", Proceedings of 5th IFAC Workshop on Algorithms and Architectures for Real-Time Control, Can-Cun, Mexico, Elsevier Science Ltd., Oxford, April, 1998, pp. 253-257
6. Ortega C. and Tyrrell A., "MUXTREE revisited: Embryonics as a Reconfiguration Strategy in Fault-Tolerant Processor Arrays", Lecture Notes in Computer Science 1478, Springer-Verlag, 1998, pp.206-217
7. Ortega C. and Tyrrell A., "Reliability Analysis of Self-Repairing Bio-Inspired Cellular Hardware", IEE Colloquium on Evolutionary Hardware Systems, London, June 1999, Digest No: 99/033
8. Ortega C. and Tyrrell A., "Biologically Inspired Fault-Tolerant Architectures for Real-Time Control Applications", Control Engineering Practice, July 1999, pp. 673-678
9. Ortega C. and Tyrrell A., "Reliability Analysis in Self-Repairing Embryonic Systems", in Stoica A. et al. (Eds.), Procs. of 1st NASA/DoD Workshop on Evolvable Hardware, Pasadena, CA, USA, IEEE Computer Society, July 1999, pp.120-128
10. Ortega C. and Tyrrell A., "Self-Repairing Multicellular Hardware: A Reliability Analysis", in Floreano D. et al. (Eds.), Advances in Artificial Life, Procs. of the 5th European Conference, ECAL'99, Lausanne, Switzerland, Lecture Notes in Artificial Intelligence 1674, Springer-Verlag, September 1999, pp.442-446

Embryonics: A Bio-inspired Fault-Tolerant Multicellular System

DPhil Thesis by César A. Ortega-Sánchez

TABLE OF CONTENTS

3. EMBRYONICS: A CONFLUENCE OF IDEAS

4. ARCHITECTURE OF AN EMBRYONIC SYSTEM

5. RELIABILITY ANALYSIS OF THE EMBRYONICS ARCHITECTURE

6. CONCLUSIONS AND FUTURE WORK

TABLE OF FIGURES

TABLES

INTRODUCTION

Electronic systems, particularly computers, have become common companions of everyday life. We can find computers in banks, at school, in our desks and even in places where they are not completely apparent like, for example, microwave ovens or vending machines. Children from new generations are growing up taking these devices for granted, for them computers have always been part of the world, computers are part of their lives. As time passes, advanced societies will more and more depend on computers for their every day functioning. Electronic money, electronic mail, virtual shopping, virtual school, teleconferencing, entertainment on-demand, travelling arrangements on-line and automated nearly autonomous manufacturing processes are options today. Tomorrow, they will be the one and only alternative.

Nowadays, computers are used not only to simplify trivial and unimportant activities. Nuclear power plants, expensive satellite and space probes, life-supporting medical equipment, fly-by-wire aeroplanes, and telephone switching systems are applications where computers are indispensable, either because of the amount of processing required or because there is no chance for human operators to participate [Avi87]. For these applications a computer failure could cost immense amounts of money, or even human lives. Therefore, for these critical applications, computers cannot stop providing their services; they should work "perfectly" for at least a predetermined period of time. This period is application dependent and can span from seconds to tens of years.

To achieve this level of availability there are two possible approaches: either to build fault-free hardware and software systems (the fault-avoidance approach), or to build hardware and software systems capable of deliver their services even in the presence of faults (the fault-tolerance approach) [Lee90]. Experience has demonstrated that the idea of building perfect systems, although attractive, is impossible to achieve. Hardware deteriorates with time, and software systems have become so complex that design faults are difficult, if not impossible, to avoid. Hence, the more viable alternative is to implement systems capable of tolerate faults, i.e. fault-tolerant systems.

All fault-tolerant systems imply the use of redundancy to achieve resilience to faults; however, the cost associated with redundancy is generally high. Cost is probably the only factor that has prevented fault-tolerant systems from being widely used, however the ratio cost/complexity of electronic systems is decreasing by the day, opening an opportunity for highly redundant systems to be extensively used.

The aim of this project is to propose a line of research where technologies from different fields of study can co-exist and give birth to a new paradigm for the design and construction of hardware fault-tolerant systems.

It is undeniable that features of biological organisms such as, for example, healing, growth, evolution and self-diagnosis would be extremely beneficial if applied to electronic circuits. In particular, fault-tolerant systems would be greatly improved if donated with such characteristics. Although direct transfer of the aforementioned biological mechanisms to silicon is impossible, recent advances in various key technologies will allow the design and implementation of bio-inspired fault-tolerant systems.

Embryonics is a nascent science that combines the latest developments in fields such as electronics, molecular biology and theory of complexity, to propose a new approach to hardware design. It departs from the observation that one of the most interesting features of biological mechanisms at cellular level is their ability to self-repair: cells are continuously killed and created. However, at higher levels of organisation, e.g. organs, limbs and bodies, the organism continues to function as if all of its original cells were still active. The basis of this robustness is the continuous replacement of old cells for brand-new cells. Cells reproduce by following a "set of instructions" stored in their DNA. These instructions, formally known as the genome of the organism, are passed from mother to daughter cells during cellular reproduction. Embryonics adopts the concept of genome and transports it to the 2-D realm of integrated circuits. The result is a family of programmable devices able to autonomously change their configuration when a fault arises in one of their components.

Hypothesis

This thesis presents the work carried out during three years of doctoral work. The aim has been to find out whether or not the following hypothesis is correct:

> ***"Embryonic systems are viable alternatives for the design and implementation of fault-tolerant systems"***

Contribution of the Research

The work reported in this thesis offers four main contributions as described below.

1. A novel approach for the design of fault-tolerant systems [Ort97a, Ort97b]. (Chapters 1, 4 and 5)
2. An alternative Field-Programmable Gate Array Architecture inspired by Biology [Ort98a, Ort98c]. (Chapters 2 and 3)
3. An original memory structure for the MUXTREE embryonic architecture [Ort98d]. (Chromosomic approach in chapter 4)
4. A novel reliability analysis that can be used to model reliability in embryonic architectures and other cellular structures as well [Ort99b]. (Chapter 5)

In addition to these main contributions, six other contributions are worthy of inclusion here:

- A survey of the field of fault tolerance from the point of view of hardware design (chapter1).
- A survey of the field of bio-inspired systems and evolvable hardware (chapter 2).
- A review of biological topics such as the embryonic development of multicellular organisms and the central dogma of Molecular Biology (chapter 3).
- A survey of Field-Programmable Gate Arrays (FPGA) technology, emphasising their internal organisation (chapter 3).
- A novel application of Ordered Binary Decision Diagrams (OBDD) and its implementation using networks of multiplexers (chapters 3 and 4).
- An original application for embryonic arrays namely, the implementation of a programmable frequency divider.

The work reported in this thesis has provided contributions for, at present, a total of ten journal, conference or colloquia papers. The complete list can be consulted on page 6.

Structure of the Thesis

Figure 0.1 shows, in a diagram, the structure of the present document.

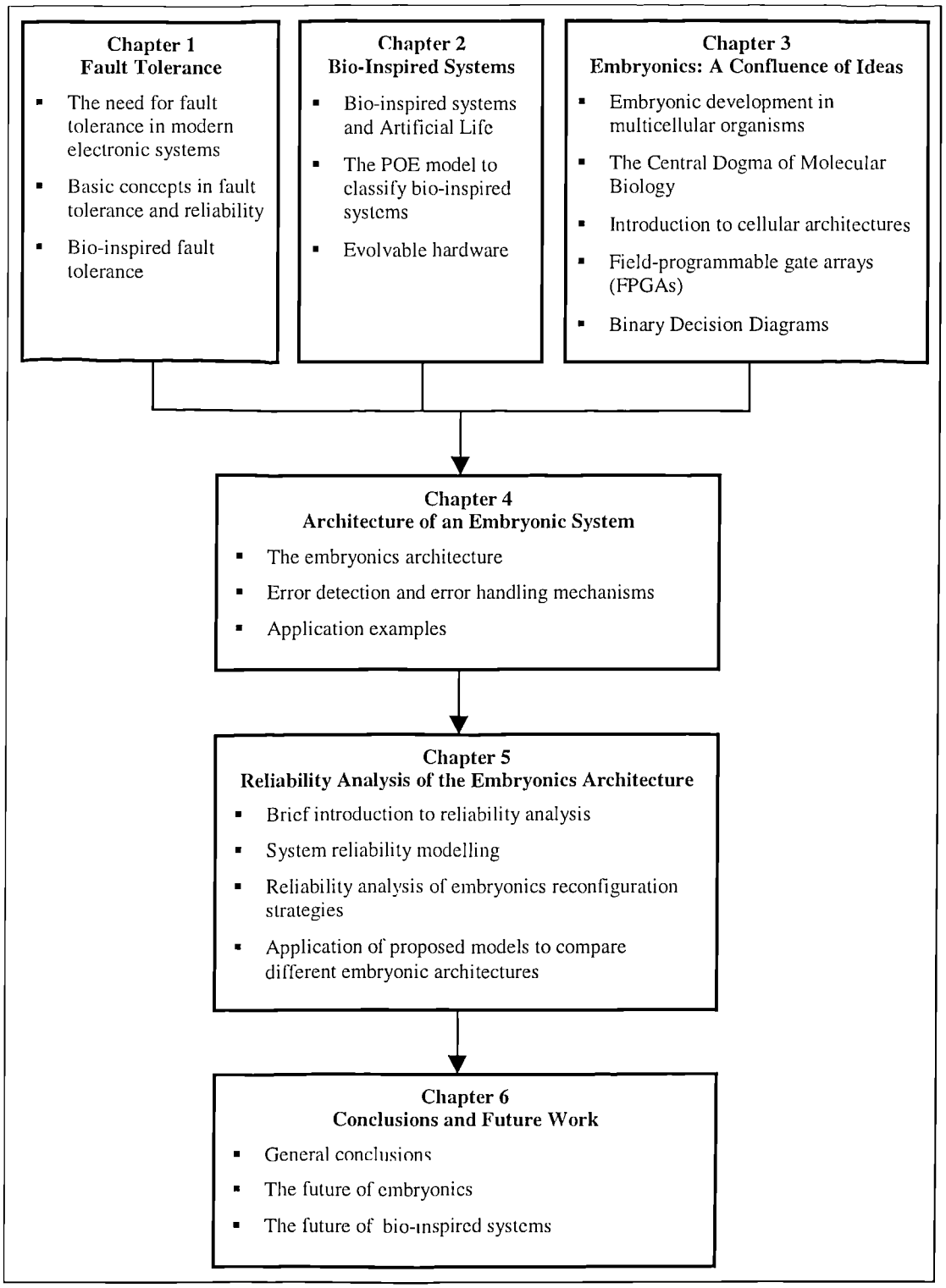


Figure 0.1 Structure of the thesis

Chapter 1 introduces the subject of fault tolerance in the context of electronic systems design. A review of the evolution of fault tolerance from its beginnings in the late 40's to the present day is given. Concepts and notations commonly used in the field of fault tolerance are defined in chapter 1. Particular emphasis is given to the problem of hardware fault tolerance. To conclude the chapter, a novel approach to solve the problem of hardware fault tolerance by drawing inspiration from mechanisms found in biological systems is proposed.

Chapter 2 is an introduction to the area of bio-inspired systems. Bio-inspired systems are defined as those systems whose design or behaviour finds a correspondent mechanism in nature. Bio-inspired systems are currently studied in the general framework of Artificial Life; hence the most relevant works in this area are briefly described in this chapter. The Phylogeny-Ontogeny-Epigenesis (POE) model for classifying bio-inspired systems is given particular attention. Chapter 2 concludes with an introduction to a particular set of electronic circuits whose behaviour is "evolved" rather than designed. It is proposed that evolvable hardware offer new alternatives to solve the problem of implementing fault-tolerant systems.

Chapter 3 gives a brief introduction to the fundamental ideas that give shape to the embryonics architecture. Embryonics is a proposal for a bio-inspired cellular architecture with inherent fault tolerance properties. The two biological fields where embryonics draw inspiration from are the development of embryos and the central dogma of molecular biology. However, other technological resources are required to implement embryonic systems, namely cellular architectures, field programmable gate arrays and ordered binary decision diagrams. A general introduction to these fields is also presented in chapter 3.

Chapters 1, 2 and 3 are self-contained and can be read in any order. The content of these three chapters provides the knowledge framework from which the embryonics architecture is derived.

Chapter 4 presents a detailed description of the MUXTREE embryonic architecture. A block diagram of a generic embryonic cell, along with a description of each one of its constituent blocks is presented. The built-in self-test techniques employed to donate the cell with fault tolerance, and the cost associated with it are discussed. The chapter concludes with three examples of the use of embryonic arrays. Resilience to faults is verified by means of simulation.

Chapter 5 provides a formal demonstration of embryonic arrays' fault tolerance. First, a review of the basics on reliability is given. Next, mathematical reliability models of well-known structures (series, parallel, k-out-of-m) and some combinations of them are presented. System reliability models for embryonics' reconfiguration strategies are obtained by

recursively applying the models for the simple structures presented before. To demonstrate that the methodology proposed can be applied to analyse other reconfiguration strategies, the reliability analysis of the MICTREE architecture is achieved. Chapter 5 concludes by demonstrating that the mathematical models proposed can be used to compare either different reconfiguration strategies or different alternatives of a particular one.

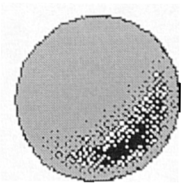
The main contributions of this work are summarised in chapter 6, along with some proposals for future work in the fields of embryonics, evolvable hardware and artificial life.

The embryonic cell was implemented using Viewlogics'® CAD suite WorkView®. The design was captured in the form of schematic diagrams. The detailed schematic diagrams of the MUXTREE cell have been included in Appendix A at the end of the document. The schematics are sufficiently detailed so as to allow the implementation of the embryonic cell in other CAD suites different to WorkView.

During the development of the research presented in this work a number of books and journal articles have been consulted. Every effort has been made to give credit where it corresponds to authors whose concepts and ideas have been used to support this research. A list with all the references cited throughout the main body of the thesis is found at the end of the thesis, after the appendix.

This research has being developed within the Bio-Inspired and Bio-Medical Engineering Group, in the Department of Electronics, University of York. Any comments or further inquiries should be addressed to:

Prof. Andy Tyrrell
Group Leader
Dept. of Electronics
University of York
York, YO10 5DD, UK
Tel: +44-(0)1904-43-2340
Fax: +44-(0)1904-43-2335
e-mail: **amt@ohm.york.ac.uk**
http://www.amp.york.ac.uk/external/media/cal/welcome.html



CHAPTER 1

FAULT TOLERANCE

This chapter presents the subject of fault tolerance in the context of electronic systems design. In section 1.2 a review of the evolution of fault tolerance from its beginnings in the late 40's to the present day is given. Section 1.3 presents some common concepts and notations used in the field of fault tolerance. Particular emphasis is given to the problem of hardware fault tolerance. Section 1.4 introduces a novel approach to solve the problem of hardware fault tolerance by drawing inspiration from mechanisms found in biological systems.

1.1 Introduction

When a system (natural or man-made) reaches a certain level of complexity, it becomes very difficult to grasp all of its underlying dynamics, and therefore, it becomes less controllable and less reliable [Pau96]. However, the needs of the modern individual are fulfilled using extremely complex systems. What would our society be without computers, satellites, aeroplanes, mega-software and free market? Complex systems are the foundation of our life-style but they have become very unreliable and difficult to design. Therefore, it is necessary to look for new methodologies and strategies to deal with complex systems. One approach is the refinement of traditional design techniques, but the techniques themselves are becoming too complex to be considered error-free. Evidently, we have to look somewhere else for answers [Avi97].

There are two fundamentally different approaches that can be taken to increase the reliability of complex computing systems. The first approach is called **fault prevention** and the second **fault tolerance**. In the traditional fault prevention approach the objective is to increase the reliability by *a priori* elimination of all faults. Since this objective is practically impossible to achieve, the goal of fault prevention is to reduce the probability of system failure to an

acceptably low value. In the fault tolerance approach, faults are expected to occur during computation, but their effects are automatically counteracted by incorporating redundancy, i.e. additional facilities, into a system, so that valid computation can continue even in the presence of faults. These facilities consist of more hardware, more software, more time, or a combination of all these; they are redundant in the sense that they could be omitted from a fault-free system without affecting its operation.

Fault tolerance is not a replacement but rather a supplement to the most important principles of reliable system design, i.e. (a) use the most reliable components and (b) keep the system as simple as possible consistent with achieving the design objectives.

The effectiveness of fault tolerance for enhancing computing system reliability is much more pronounced in a system composed of basically reliable components than in a system of unreliable components. In other words, while fault tolerance can be used to increase significantly the reliability of an already reliable system, it is of little use, and can even have a detrimental effect, if the original system is unreliable in the first place.

This chapter presents a historical review of fault tolerance since its beginnings, and a revision of the techniques currently available. The present chapter provides the conceptual framework upon which the remaining chapters of this thesis are sustained.

1.2 Evolution of Fault Tolerance

Fault tolerance is not a new idea. The first digital computers made extensive use of error detection and fault tolerance techniques to overcome the low reliability of their basic components. Some of the early Bell Relay Computers (BRC), for example, had two central processing units; one unit would begin executing the next instruction when the other unit encountered an error [Pro48]. Later versions of the BRC used a retry mechanism to repeat an operation immediately after an error was detected. The IBM 650, UNIVAC, and the Whirlwind I computers incorporated parity to check the results of data transfers. The EDVAC computer designed in 1949, is generally considered to have been the first computer to completely duplicate the Arithmetic Logic Unit (ALU) and compare the results obtained by each unit; the processing continued as long as the two ALUs agreed.

The advent of the transistor, along with its increased reliability, led to a temporary decrease in the emphasis on fault-tolerant computing. For many designers, the major thrust was to increase computer performance and speed and to depend on the improved reliability of the transistor to guarantee correct computations. It was not until computers began performing much more critical tasks that fault tolerance again surfaced as a crucial issue [Avi78].

The first theoretical work in fault-tolerant computing is generally credited to John von Neumann. In 1956 von Neumann wrote an article entitled "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components", in which he presented the concept of majority voting and analysed the impact that such arrangements could have on the probability of a system producing erroneous results [Neu56].

Since about 1970, the field of fault tolerant computing has been rapidly developing. Several journals such as Computer, IEEE Micro, the Proceedings of the IEEE, the Journal of Design Automation and Fault Tolerant Computing, and the IEEE Transactions on Computers regularly present special issues that deal solely with fault-tolerant computing.

In addition, the International Symposium on Fault-Tolerant Computing (FTCS) has been held each year since 1971. Its proceedings present the results obtained by researchers in industry, academic institutions and government laboratories all around the world. The topics covered include reliability modelling, architectural concepts, fault detection methodology, and recovery techniques among others.

In recent times, some research has been focused on the fault tolerant attributes of massively parallel processing element networks, such as artificial neural networks. In this approach the "knowledge" is distributed throughout the multiple processing elements, therefore, if one or a relatively small part of the processors fails, the overall functionality could be maintained [Bar89] [Che90] [Dye95] [Gar96].

1.3 Basic concepts and definitions

As the community of Fault Tolerance researchers is growing all around the world, a common vocabulary becomes necessary. In [Lap92] Laprie proposes informal but precise definitions characterising the various attributes of computing systems dependability. The majority of the scientific and technical community working on reliable and fault tolerant computing has accepted this nomenclature.

In [Lap92] dependability is defined as the trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers. For different users (human or physical), the concept of dependability can vary depending on the properties of the service delivered by the system:

With respect to the readiness for usage dependable means available.

With respect to the continuity of service dependable means reliable.

With respect to the avoidance of catastrophic consequences dependable means safe.

With respect to the prevention of unauthorised access and/or handling of information, dependable means secure.

A system failure occurs when the delivered service no longer complies with the specification. The specification is an agreed description of the system's expected function and/or service. An error is that part of the system's state which is liable to lead to subsequent failure. The adjudged or hypothesised cause of an error is a fault.

Summarising: A fault leads to an error, which leads to a failure.

The development of a dependable computing system calls for the combined utilisation of a set of methods. These methods can be classed into:

Fault prevention: how to prevent fault occurrence or introduction.

Fault tolerance: how to provide a service complying with the specification in spite of faults.

Fault removal: how to reduce the presence of faults.

Fault forecasting: how to estimate the present number, the future incidence, and the consequences of faults.

Fault prevention and fault tolerance may be seen as constituting dependability procurement, i.e. how to provide the system with the ability to deliver a service complying with the specification. Fault removal and fault forecasting may be seen as constituting dependability validation, i.e. how to reach confidence in the system's ability to deliver a service complying with the specification. Figure 1.1 shows the dependability tree. The dependability tree summarises the concepts introduced up to now [Lap92].

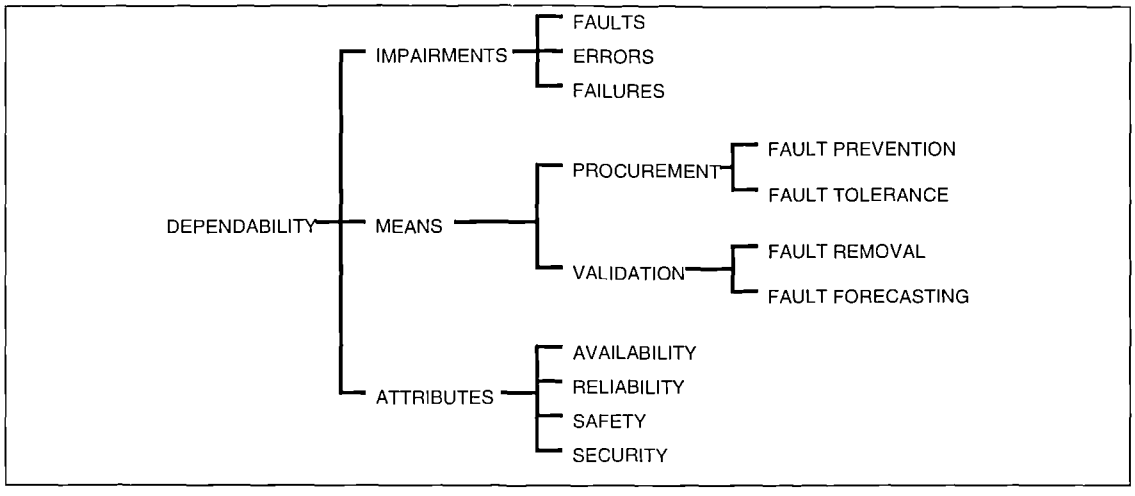


Figure 1.1 The dependability tree

Faults and their sources are extremely diverse. They can be classified according to three main viewpoints: their nature, their origin and their persistence. Figure 1.2 summarises the classification of faults according to [Lap92].

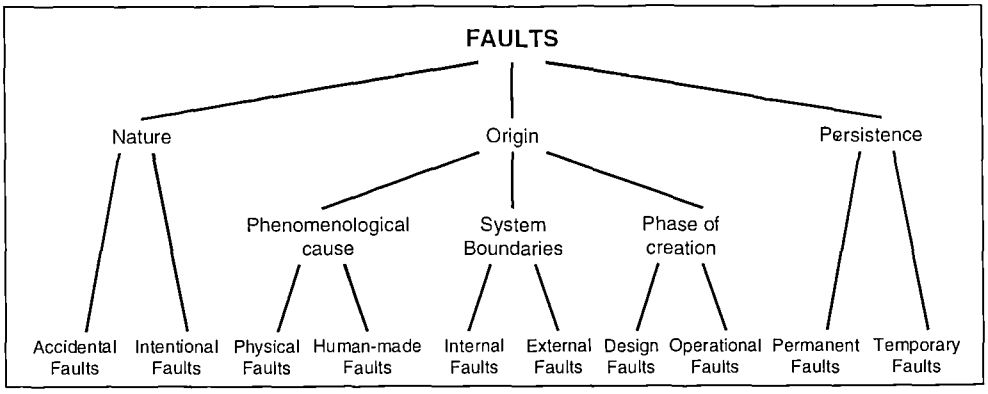


Figure 1.2 Classification of Faults

In the previous discussion the notion of system has been taken for granted, but a more rigorous definition must be given in order to understand the following sections.

A system is any identifiable mechanism that maintains a pattern of behaviour at an interface between the mechanism and its environment. An interface is simply a place of interaction between two systems. The environment is another system that receives a service from the first system. The external behaviour can be described in terms of a finite set of states. The specification must clearly indicate the valid states of a system. A failure is said to occur when the system reaches a state that was not considered by the specification.

From the internal structure point of view, a system is defined to consist of a set of components that interact under the control of a design. A component of a system is another system. This recursion continues up to the point when a system whose internal structure cannot be discerned or is not of interest if reached.

The internal state of a system is defined to be the ordered set of the external states of its components. The design defines and controls system's parts interaction and the flow of inputs and outputs into and from subsystems.

Imposing structure is the basis for controlling complexity, and hence it is the basis of methodologies for designing and constructing both hardware and software for reliable computing systems [Lee90].

1.3.1 The phases of fault tolerance

Given the impossibility of designing fault-free systems, it is necessary to incorporate fault tolerance to improve their reliability. Fault tolerance techniques, either software or hardware, always imply the use of **redundancy**. Redundancy can be either **static** or **dynamic**. In the first case redundant components are operative all the time, and the result delivered by the system is a function of the results given by both the main and redundant components. In the second case, spare components are kept inactive until a fail in the active component is detected, then one of the spares (there can be several) is activated and updated with the most recent valid system information in order to substitute the failing one. In general, fault-tolerant system should be able to implement the following phases [Lee90]:

i) Error detection.

ii) Damage confinement and assessment.

iii) Error recovery

iv) Fault treatment and continued system service

In practice there can be considerable interplay between the various phases, which tends to blur their identification in a particular system. Phases ii), iii) and iv) can be used in any order depending on the system, and nor necessarily must all three phases be present. This is the case when, for example, the repair of faults relies on manual intervention.

Error detection

In this phase the objective is to detect the presence of errors before they propagate throughout the system and provoke a failure. Once the error is detected, either an exception can be signalled so that other parts of the system can handle it, or the error is handled or masked in the same module that detected it. In theory, if all errors were detected and handled, no failure would occur, at the expense perhaps, of some system performance deterioration. But to achieve this level of error manipulation is impractical due to cost limitations and overheads.

There are a variety of error-checking techniques that can be applied; some of them are described next. [Lee90]

Replication checks.- Probably the best technique but the most expensive. It detects possible errors better than any other technique. This technique implies the complete replication of the system whose state is being monitored. It follows the static redundancy approach. For error

detection only two replications are needed. To mask errors, Triple Module Redundancy (TMR) or N-Module Redundancy (NMR) is needed.

Timing checks.- Uses "time-outs" to detect the presence of a fault, but not its absence. Watchdog timers are used in hardware and/or software to warrant that the system will not stuck on an infinite loop. This technique is widely used in real-time applications.

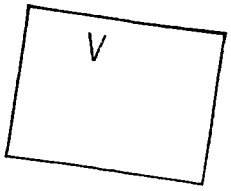
Reversal Checks.- Used when the relationship between input and output is one-to-one. The basic principle is to recalculate the input from the output and compare the result with the actual input. The main problem with this technique is the overhead imposed to recalculate the input. This process often takes more time to be completed than the original calculation. Reversal checks must be an independent part of the system in its design and implementation.

Coding Checks.- Are based on redundancy in the representation of an object in use in a system. Non redundant data are associated to the redundant data in a fixed relationship that must be kept. Examples of the most commonly used coding techniques are: Parity checks, Hamming codes, M-out-of-N codes, cyclic redundancy codes, signature instruction streams and arithmetic codes. Coding checks need little redundancy but are at best a limited form of acceptability check.

Reasonableness Checks.- Based on a knowledge of the internal design and construction of the system. Applicable when the complete set of valid outputs is known. Examples: Range checks, change-rate check, consistence with other objects in the system, type checking in software. Explicit checks for reasonableness included in software systems are sometimes termed **assertions** (assert statements), they evaluate to false if the state is erroneous. Run-time checks in software add too much overhead to the system. They are included only during testing.

Structural Checks.- Applied to data structures in software systems. They can check semantic integrity or structural integrity. Structural checks are particularly applicable to lists, queues and trees. Examples: Duplicated pointers, information elements in the structure.

Diagnostic checks.- Concerning specifically with checking the behaviour of the components of which the system is constructed. These kinds of checks exercise the component with inputs and check the outputs. They are applied periodically (at start-up time) or in the background because of their expensiveness in terms of time and resources required for their execution.



Damage confinement and assessment

To protect critical system resources and minimise recovery time, errors must be confined to the module in which they originate. Typically, error-containment boundaries are hierarchically defined, with errors confined at the lowest level to single replaceable or repairable modules, and additional boundaries set around subsystems containing this modules. Containment boundaries can be established in two ways: Each module can check its own outputs, or each can check all incoming information. The most common approach is to require each module to suspect all incoming information and correct or contain faulty data at the module interface. Voters are extensively used for this purpose.

If a module is to be responsible for its own output, it needs an error-containment boundary. An error detection or correction circuit, such as a voter, a comparator, or a code checker, is placed at the interface between the module and the system bus or communications channel, along with a circuit capable of disabling the module's output. If error correction is not possible, a faulty module must be isolated to prevent error propagation. A disadvantage in this configuration is that the module interface often cannot protect the system from failures of the interface circuits themselves [Nel90].

Strategies for damage assessment rely on the structure that the designer assumes will be present in the operational system. Hence, damage assessment (often involving subjective decisions) will be system specific.

Error recovery

The damage an error can generate could be anticipated or unanticipated. **Forward** and **Backward Error Recovery** are the respective techniques applied to recover a system once error have been detected.

Forward error recovery is always system specific and the success of this approach depends upon the accuracy with which damage can be predicted (and assessed). Redundant data and forward error recovery form the basis of error correcting codes which are used to recover from faults in memory units. Error correcting codes utilise redundancy to enable the position of the erroneous bit(s) to be calculated.

When backward error recovery is applied, the entire state of the system is replaced by a prior state known as not erroneous in an attempt to simulate the reversal of time. The replacement of the entire state of a system is called a reset of the system. The most basic reset is to place the system in some predefined state (initial). This kind of reset is called "cold start". If the reset can be done to several states different from the initial, this reset is referred to as "warm

reset". If the system can be restored to a state which it occupied prior to the manifestation of a fault, and if the fault was a temporary one, then all errors resulting from that fault must have been removed.

A flexible backward error recovery facility will permit more than one recovery point to be available; hence recovery data must be available for each of them. A **recovery point** is a fault free state to which the system is taken when errors occur. A recovery point is said to be active from the time at which it is established until it is discarded. That is called the **recovery region** for the recovery point.

Fault treatment and restoration of service

To ensure reliability it is not enough to remove errors and return the system to a safe state, it is also necessary to eradicate those faults that provoked the errors; otherwise errors can manifest over and over again.

Fault treatment techniques commonly are system specific and difficult to implement, in fact so difficult that sometimes the minimal approach is taken: Ignore the fault and hope for the best [Lee90]. This approach could be successful if:

1. Error recovery is powerful enough to cope with recurring faults.
2. Future operation of the system fortuitously avoids the fault
3. The fault is transient.

Hardware faults are often transient, but software faults are always permanent because they are design faults. Hence, for software systems without variants (redundant software modules designed independently from the original program), backward error recovery will not work since it would be futile to restore a prior state and try again with exactly the same program.

In general, fault-treatment techniques require two stages: fault location and system repair.

Fault Location

Automatic repair of the system will only be possible if the failure exception provides an accurate guide to the location of the fault. The most important exploratory technique used to locate faults is that of **diagnostic checking,** either in hardware or in software. In diagnostic checking a component is invoked with a set of inputs for which the correct outputs are known. Fault location usually precedes system repair, but a pessimistic and cautious alternative defers fault location until system repair is under way. In this pessimistic approach

all but a small set of trusted components are assumed to be faulty. Only components that pass the diagnostic checks are accepted as not being faulty.

System repair

Techniques for system repair are based on some reconfiguration of the system so that the characteristics of use of suspect components are modified to some extent. Reconfiguration techniques have been classified as [Lee90]:

- **Manual**: All actions are performed by an agent external to the system (usually human)
- **Dynamic**: Actions are performed by the system in response to instructions from its environment.
- **Spontaneous**: All actions are initiated and performed by the system itself.

Dynamic and spontaneous techniques are only found in inaccessible or highly available systems. These techniques use switching networks to reconfigure interconnections or components. Components suspected of being faulty are replaced by stand-by spares. In hardware systems identical designs are used. In software systems different designs are needed because all faults are design faults. As mentioned before, spares do not necessarily have to be idle, they can be used to do some work and the elimination of any module results in a **graceful degradation** in the standard of service provided.

The size of replaceable components is important. Large components are easier to be detected faulty but they impose large redundancy overhead on the system. Small components have lower MTBF but the switching network to interconnect them becomes to complex. Dynamic reconfiguration is preferred over spontaneous because sometimes faults cannot be located automatically and if they can, manual confirmation must be given before the system is allowed to reconfigure.

Resuming Normal Service

If recovery can be achieved by means of a fixed reset then this technique would probably be adequate. For hardware systems a retry is frequently attempted. Retry is the cheapest form of redundancy in every commodity except time. In software systems there is flexibility of action. Control can be transferred to an appropriate location or the exception handler should terminate by signalling a failure exception.

1.3.2 Software Fault Tolerance

In general terms, the major proportion of complexity of most systems is to be found in the software. There are two main methods to provide software fault tolerance: Recovery Blocks [Hor74] and N-version programming [Lap90]. Neither of these provides an absolute guarantee that the fault tolerance provided will be successful.

The Recovery Block Scheme

For this technique there is a software module designed and tested to satisfy a specification - **the primary module-**, but it is likely to contain design faults. In the event of primary module failing an **alternate module,** sometimes referred to as a **variant** [Lap90], will be used as a stand-by spare. The alternate module must have a different design so that it will not suffer from the same fault. There can be multiple stand-by spares, or even nested ones.

The N-Version Programming Scheme

For this technique N variants are executed and its results compared. A **voter** eliminates erroneous results and pass on the (presumed to be correct) results generated by the majority to the rest of the system.

To implement N-version programming a driver program is needed to control the N versions. This driver invokes each of the versions, waits for the versions to complete its execution and compares and acts upon the N sets of results to give an output.

When results are integers or textual sets, exact agreement can be expected. When using floating point, inexact voting is required, e.g. averaging, thresholding, ignoring bits. Voters can be classified as follows [Bas95]:

- Majority voters- Agreement if (N+1)/2 variants give similar results.
- Plurality voters- Agreement if 2-out-of-N results are similar.
- Median voters- Takes the result closest to the median.
- Weighted averaging voters- Increases differences between results using weights.

The first two are voters that only generate an output if agreement is reached among variant's results. The other two always deliver a result no matter how deviant the results are.

1.3.3 Hardware Fault Tolerance

Essentially all modern hardware fault-tolerant systems achieve fault tolerance by using redundant components in one form or another. Error detection and recovery can be performed either by an external element (e.g. central processor or dedicated hardware), or by the system itself depending on the complexity of the modules to be replicated and the level of fault tolerance required. However, a central agent constitutes a single point of failure for the system and therefore, in ultra-high reliability applications, self-testable and self-reconfigurable modules should be preferred.

Retry Strategy

It has statistically been shown that the majority of hardware faults are transient [Tas77]. Cross talk, electromagnetic discharges, instant variations in the power supply and α-particle radiation are some of the phenomena that can provoke a temporary malfunction in electronic equipment. Hence, the simplest technique to achieve fault tolerance in hardware is to repeat the operation that was detected in error. Redundancy is needed only to detect the fault; neither error masking nor error-correction is achieved. However, if a higher level of dependability is expected from the system, a different technique must be used.

Backup Computers

The earliest form of hardware fault tolerance was for the computing centre to provide a complete backup or **spare system**, including memory, CPU and I/O processors. In case of computer failure, personnel transferred all work to the backup system, which then took over until repair personnel could fix the main system. This form of redundancy was the standard for critical operations, such as military defence and space exploration systems [Bar92].

Reconfigurable duplication, similar to the backup computer technique except that it occurs at the component level, is the ability of a system with redundant components to reconfigure itself dynamically. Components are duplicated and their results compared in a separate circuit. When the results do not match, the comparator generates an error signal. The operating system then determines which component failed and uses the other. Notice that the comparator and switch are critical components in this configuration.

Watchdog Timers and Heartbeats

When two or more systems operate concurrently, each one needs some way of notifying the others that it is still functioning. One mechanism is for each process to periodically notify the others that it is operational; such notification is called a **heartbeat**. If a processor does not

receive a heartbeat from another processor when it expects to, the first assumes the second has failed and operates accordingly. For example, if two processors are designated main processor and backup and the backup does not receive a heartbeat from the main processor, the backup then takes control and operates as the main processor.

Pair-and-Spare

The **pair-and-spare strategy** uses redundancy both for error detection and reconfigurable duplication at the system level. At the component level the designer uses a pair of identical components to build a unit that detects its own errors. The two components receive exactly the same inputs and simultaneously perform the same operation on those inputs. Comparison circuitry checks the outputs and generates an error signal if a mismatch occurs.

At the system level of organisation, the designer builds a computer using a pair of the error-detecting units just described. One pair operates as the main unit and the other as a spare unit. Thus there are four copies of the system components. In general, each main unit and spare unit operate in a tightly synchronised mode. If either the main unit or its spare generates an error signal, a control unit disables it and automatically switches operation to the spare if the main unit fails, so the computer continues to operate using the functional unit as the main unit. After the faulty unit has been repaired, the system brings both units into synchronous operation again.

N-Modular Redundancy with Voting

Pair-and-spare logic pairs two identical components as a way of detecting faults. **N-modular redundancy** is similar, but with N components ($N \geq 3$). Special voting logic compares the outputs and accepts the majority output as correct. Thus the system not only detects an error but also masks it. Units of this type are particularly useful in systems that cannot be repaired, such as on-board computers for guidance control [Bar92].

Built-In Self-Test

Advances in VLSI technology have led to the fabrication of chips that contain a very large number of transistors. The task of testing such a chip to verify correct functionality is extremely complex and often very time consuming. In addition to the problem of testing the chips themselves, the incorporation of these into systems has caused the cost of test generation to grow exponentially [Lal97]. For example, the approximate cost of detecting a fault at the board level is 10 times as high as detecting a fault at the chip level, and the cost increases by about 10-fold from board level to system level [Wil86].

A widely accepted approach to deal with the testing problem at the chip level is to incorporate **built-in self-test (BIST)** capability inside a chip. The internal state of a circuit incorporating BIST logic is continuously monitored, making the test generation and fault detection easier.

Ideally, a BIST scheme should be easy to implement and must provide high fault coverage [Lal97]. One way of achieving self-checking design is by means of error-detecting codes, where for every valid system input there is a valid code associated to the corresponding output. A code checker detects the presence of fails when its input is not a member of the set of valid codes. Figure 1.3 shows the block diagram of a totally self-checking circuit. It consists of a functional circuit and a checker, both are supposed to be totally self-checking.

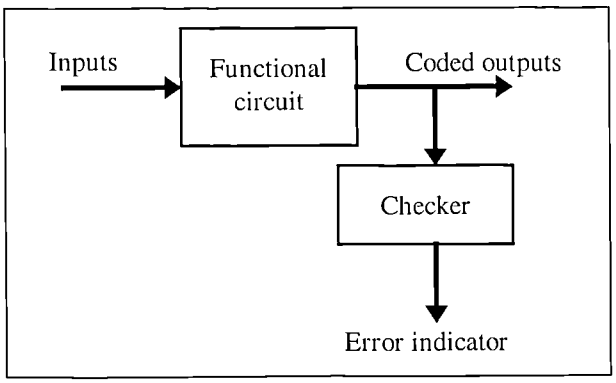


Figure 1.3 Totally self-checking circuit

By observing the output of the checker it is possible to detect any fault in the functional circuit or the checker itself. A totally self-checking checker must have two output lines and, hence, four output combinations. Two of these output combinations are considered as valid, namely (01,10). By choosing these combinations where both bits change their value, it is easy to detect faults that stuck these lines to either logic zero or one. A non-valid checker output, 00 or 11, indicates either a non-code word at the input of the checker or a fault in the checker itself [Lal85].

1.4 Bio-Inspired Fault-Tolerance

Nature offers to us some remarkable examples of how to deal with complexity and its associated unreliability. For example, the human body is one of the most complex systems ever known. Local failures are common, but the overall function of our organism is highly reliable because of the self-diagnosis and self-healing mechanisms that work ceaselessly throughout our bodies. These mechanisms are the result of millions of years of our genes'

evolution. Evolving instead of designing seems to be an attractive alternative when dealing with complexity [San96a].

During the past few years the work done on bio-inspired systems has generated some remarkable results [San96a, Hig97, Sto99]. Genetic algorithms, neural networks, artificial brains and evolvable hardware are just a few examples of this novel approach. What is in nature that is so attractive for hundreds of engineers and scientists? The answer can be found in the characteristics that biological organisms possess. Characteristics such as evolvability, multi-cellular structures, auto-regulation, and learning that allow them to adapt to changes in their environment.

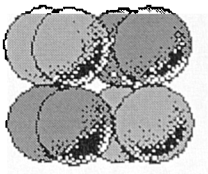
A recent approach to fault tolerance is that of borrowing from nature the main principles that make living beings so resilient to faults. Mechanisms such as self-diagnosis, self-healing, reproduction and adaptation are being transported to the arena of Computer Science and Electronics. All these characteristics seem to be a natural consequence of the massively parallel arrays of cells that constitute every living being. In the following chapters the design of a fault tolerant cellular architecture inspired by biological processes will be presented.

1.5 Summary

Modern societies rely on computers and electronic systems for their correct functioning. As these systems grow and become more complex their reliability tends to decrease. To cope with the fact that all systems will eventually fail, fault tolerance has to be incorporated in our designs. Incorporating fault tolerance into a system implies the use of redundancy with its associated cost. However, recent developments in VLSI manufacturing conjugated with the constant drop in semiconductor prices, are allowing fault tolerance to be again an alternative.

This chapter presented the evolution of fault tolerance, from the early days after the Second World War to the present day. Also, the main concepts and techniques used to donate software and hardware systems with fault tolerant properties have been exposed. They constitute the conceptual framework over which the remaining of this thesis is sustained.

CHAPTER 2



BIO-INSPIRED SYSTEMS

This chapter is an introduction to the area of bio-inspired systems, i.e. systems whose design or behaviour finds a correspondent mechanism in nature. Section 2.2 presents Artificial Life as the general framework to study bio-inspired systems. Section 2.3 introduces the POE model for classifying bio-inspired systems. Section 2.4 presents a particular set of electronic circuits whose behaviour is "evolved" rather than designed. Evolvable hardware offers new alternatives to solve the problem of implementing fault-tolerant systems.

2. 1 Introduction

In chapter 1 it was argued that fault tolerance is a necessary feature of modern electronic equipment. It was concluded that drawing inspiration from nature could generate a new paradigm for the design of fault-tolerant systems. This chapter presents **Artificial Life** (ALife) and **bio-inspired systems** as the scientific disciplines that study man-made systems whose behaviour reminds of processes typically found in nature.

Section 2.2 introduces the main ideas sustaining bio-inspired systems and Artificial Life. It is argued that although the interest in constructing machines able to mimic biological characteristics has been present since ancient times, the technology available at that time did not allow any realistic implementation. Nowadays computers and their ever-growing processing power allow the simulation of bio-inspired systems. Future technologies might allow, in the not-so-far future, the physical implementation of artificial living beings.

Section 2.3 presents the POE model as a classification framework for bio-inspired systems. It is shown that by evolving, growing and learning, adaptation to changes in the environment and hence, fault tolerance, can be achieved.

Section 2.4 introduces **evolvable hardware** (**EHW**); i.e. hardware able to autonomously adapt to changes in its environment. This feature might turn EHW into the ideal physical substrate of future ALife implementations.

2. 2 Bio-Inspired Systems and Artificial Life

Living beings have always inspired the imagination of inventors and scientists throughout history. For example, according to Greek mythology, Daedalus invented a pair of wings made out of waxed feathers so that he and his son Icarus could escape from the labyrinth of Crete. They succeed, but Icarus flew so close to the sun that the wax melted and his wings dismantled, with tragic consequences. In renaissance Italy, Leonardo da Vinci invented flying machines and submarines inspired by birds and fish more than 500 years ago, but his inventions never came into real practice. These attempts failed not because there was something wrong with the design itself, but mainly because the available technology was far too primitive for any realistic implementation of such projects.

During the past few years, we have been witnesses to a merging of innovative ideas with powerful technologies, breathing life into the old dream of constructing machines able to mimic some of the mechanisms that make inanimate matter come alive. This topic, in its modern form, was first raised almost fifty years ago, during the post war era, by the founding fathers of cybernetics, most notably John von Neumann. Central to his final work were the concepts of self-reproduction and self-repair; unfortunately, the technology available at the time was far removed from that necessary to implement his ideas [Neu66].

The years that followed have seen the rise, fall, and eventual resurgence of artificial neural networks, along with the recent advent of artificial life, spearheaded by Christopher Langton [Lan89]. Central to artificial life research is the application of mechanisms that sustain natural evolution to artificial systems. Pioneered most notably by John Holland, this concept is slowly making headway, finding its place in the more traditional engineering disciplines as well as within the artificial intelligence community [Hol92].

The remarkable increase in computational power and, more recently, the appearance of a new generation of programmable logic devices, have made it possible to put into actual use models of genetic encoding and artificial evolution. This has led to the simulation and ultimately the hardware implementation of a new brand of machines. We have crossed a technological barrier, beyond which we no longer need content ourselves with traditional approaches to engineering design; rather, we can now evolve machines to attain the desired behaviour. This novel approach has been quite appropriately named evolutionary

engineering: "The art of using evolutionary algorithms to build complex systems" [Gar96]. Although we are just taking our first steps, it promises to revolutionise the way we will design our future machines; we are witnessing the nascence of a new era, in which the terms 'adaptation' and design will no longer represent opposing concepts [San96a].

The term Artificial Life literally means, "life made by humans rather than by nature" [Lan95]. Natural evolution implies populations of individuals, each possessing a description of their physical features, the **genotype**. A new generation of individuals is created through the process of reproduction, in which genotypes are transmitted to the descendants, with modifications due to crossover and mutation. These genetic operations take place in an autonomous manner within each entity, that is, within the genotype; the resulting physical manifestation of an individual, known as the **phenotype**, is then subjected to the surrounding environment, which, through a culling process, preserves only the best-adapted individuals. The evolutionary process has neither a central controller nor an ultimate goal toward which it strives; an individual's fitness is implicitly determined by its ability to survive and reproduce in the surrounding environment [San96a].

Natural life on earth is organised into at least four fundamental levels of structure [Tay95]:

- Population-Ecosystem level
- Organism level
- Cellular level
- Molecular level

Understanding life in any depth requires knowledge at all the four levels. Biological sciences are using artificial life systems to understand natural life. Hardware systems are used to study the organism level. Cellular and population levels are studied through the use of software systems. The molecular level is studied through experiments with RNA molecules (Wetware).

Artificial life work can be divided into the design of systems with "biological properties" to accomplish a particular task, artificial neural networks for example; and systems meant to accurately model biological systems in order to test biological hypotheses, like genetic engineering. Research on Artificial Intelligence has been carried out in Computer Science laboratories for more than forty years; but research in Artificial Life is more modern and still in its infancy. Therefore, it is important to make a clear distinction between artificial intelligence and this novel paradigm called artificial life. Table 2.1 presents some important differences between the two disciplines [Dye95]:

Artificial Intelligence (AI)	Artificial Life (ALife)
- Focus on individuals	- Focus on a group or population
- Cognition as operations of logic	- Cognition as operation of nervous systems
- Cognition independent of perception	- Situated cognition with sensory/motor experiences
- Starts with human level cognition	- Starts with animal level cognition
- Mainly top-down approach: Engineer complex systems	- Mainly bottom-up approach: rely on evolution, development and learning
- Direct specification of cognitive architecture	- Indirect specification of cognitive architecture via genotype to phenotype mapping
- Human level mental tasks	- Survivability in complex environments is the overriding task
- Time span up to hours	- Evolutionary, generational and individual life spans.

Table 2.1 Differences between Artificial Intelligence and Artificial Life.

A very long-term goal of Artificial Life is to gain insight ultimately into the evolution and nature of human intelligence, through modelling the evolution of communication and co-operative behaviour in lower life forms.

The ALife modelling approach involves the specification of:

1. **Environments**.- Simulated worlds whose conditions match, at some level of abstraction, those selecting pressures in which a variety of animal behaviours may evolve or develop.
2. **Processes of genetic expression**.- Mapping from artificial genomes to phenotypes that control behaviour.
3. **Learning and development**.- Methods under genetic control for modifying or growing the nervous systems of artificial animals during their lifetimes.
4. **Evolution**.- Recombination and mutation of parental genomes to produce variation in their offspring.

It is important to keep in mind while designing any form of artificial life that it took only 1 billion years or so for the first cells to form on earth, but about 3 billion more years for these to evolve into metazoans (multicellular organisms). Hence, success in any form of life, natural or artificial will require, most certainly, long periods of time.

One distinctive characteristic of bio-inspired systems is that of their complexity. Systems are so complex that a complete description or design of their functionality is practically impossible. Instead, a rudimentary quasi-random configuration is encouraged to evolve and, after some iteration with the environment, some kind of order begins to appear from the original configuration. Apparently, that has been the secret of natural life success [Kau96].

2. 3 The POE Model

If one considers life on earth since its very beginning, then three levels of organisation can be distinguished [Man98]: Phylogeny (P), Ontogeny (O), and Epigenesis (E). In analogy to nature, the space of bio-inspired systems can be partitioned along these three axes. This is called the POE model [Sip97b]. Figure 2.1 shows the model as three orthogonal axes.

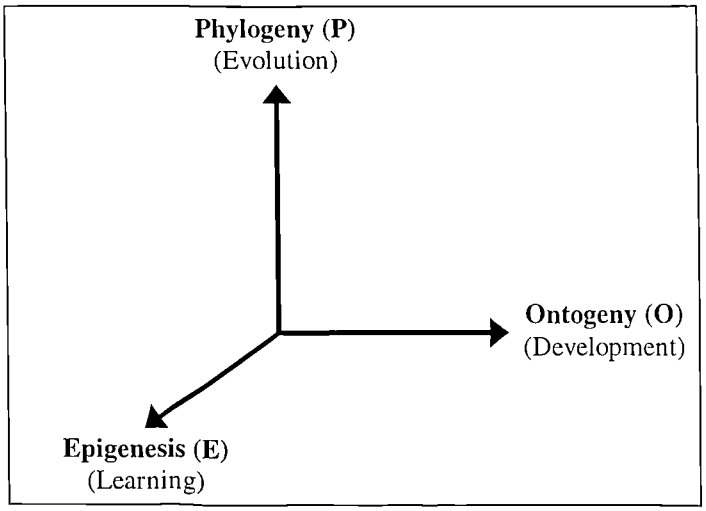


Figure 2.1 The POE model and its associated adaptive processes

For example, consider the following three paradigms, each one positioned along one axis:

- Evolutionary algorithms are the (simplified) artificial counterpart of Phylogeny (P) in nature.
- Self-reproducing automata are based on the process of Ontogeny (O), where a single mother cell gives rise, through multiple divisions, to a multicellular organism.
- Artificial neural networks embody the Epigenesis (E) process, where the system's synaptic weights and perhaps topological structure change through interactions with the environment.

Within the domains collectively referred to as soft computing, characterised by ill-defined problems coupled with the need for continual adaptation or evolution, the above paradigms yield impressive results, rivalling those of traditional methods.

2.3.1 Phylogeny

Phylogeny concerns the temporal evolution of genetic programs (genomes). The hallmark of phylogeny is the evolution of species. The "multiplication" of living beings is based upon the reproduction of the genome, subject to an extremely low error rate at the individual level, so

as to ensure that the identity of the offspring remains practically unchanged. This error rate is higher at the group or species level. It is precisely these copying errors, due to mutation (asexual reproduction) or mutation along with recombination (sexual reproduction), that gives rise to the emergence of novel species or new organisms. The phylogenetic mechanisms are fundamentally non-deterministic, with the mutation and recombination rate providing a major source of diversity. Diversity is indispensable for the survival of living species, for their continuous adaptation to a changing environment, and for the appearance of new species.

The phylogenetic axis admits a number of qualitative sub-divisions, where different implementations of the paradigm can be accommodated This is shown in figure 2.2 [San97].

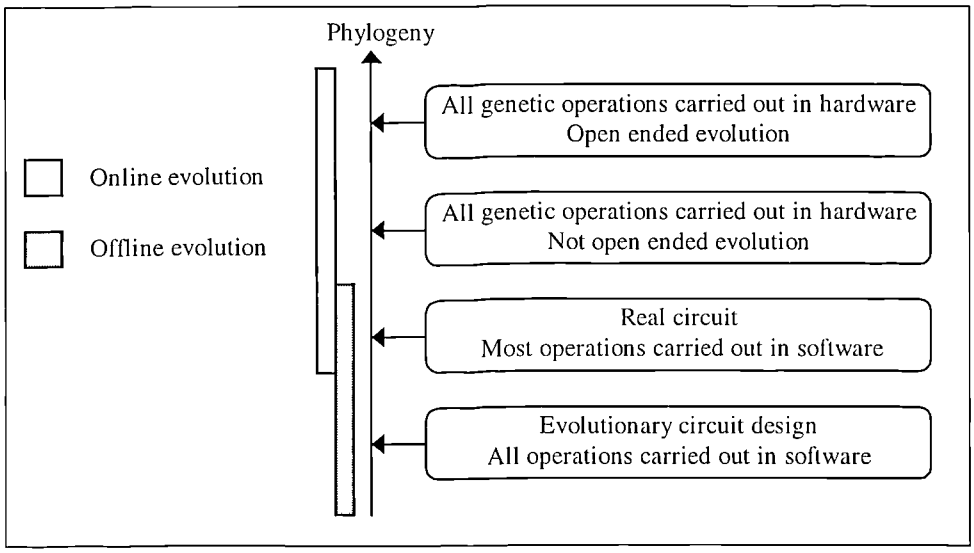


Figure 2.2 The phylogenetic axis

At the bottom of this axis, we find what is in essence *evolutionary circuit design*, where all operations are carried out in software, with the resulting solution possibly loaded onto a real circuit. Though a potentially useful design methodology, this falls completely within the realm of traditional evolutionary techniques. As examples one can cite the works of [Hem96], [Hig96] and [Kit96].

Moving upward along the axis, one finds work in which a real circuit is used during the evolutionary process, though most operations are still carried out offline, in software. An example is the work of [Tho96b], where fitness calculation is carried out on a real circuit.

Still further along the phylogenetic axis, one finds systems in which all genetic operations (selection, crossover, mutation and fitness evaluation) are carried out online, in hardware. The major aspect missing for a completely evolvable hardware concerns the fact that

evolution is not open-ended, i.e., there is a predefined goal and no dynamic environment. An example is the work of [Goe97].

The top of the phylogenetic axis represents a population of hardware entities evolving in an open-ended environment. There is no work reported at this level of the axis (yet!).

2.3.2 Ontogeny

Upon the appearance of multicellular organisms, a second level of biological organisation manifests itself. When biological multicellular organisms reproduce, the new individual is formed out of a single cell (the fertilised egg). During the weeks that follow the time of conception, the egg divides itself by a mechanism called mitosis. The result of mitosis is two cells with identical genetic material (DNA). The new cells continuously repeat mitosis, passing to every offspring a complete copy of its DNA. During this reproductive process cells differentiate to shape the tissues, organs and limbs that characterise a complete healthy individual of a particular species. Differentiation takes place according to "instructions" stored in the DNA (bio-chemical medium containing the genome). During differentiation different parts of the DNA (genes) are interpreted depending on the position of the cell within the embryo. Before differentiation cells are (theoretically) able to take over any function within the body because each one possess a complete copy of the DNA. Ontogeny is therefore the developmental process of a multicellular organism; this process is essentially deterministic: an error in a single base within the genome can provoke an ontogenetic sequence that results in notable, possibly lethal, malformations. The fundamental principle of embryology in real life is illustrated in figure 2.3, which covers a period of two generations preceded and followed by an indefinite number of generations [San97].

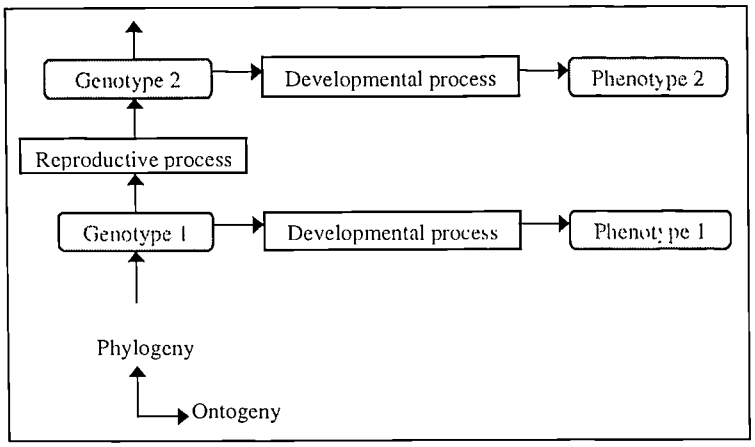


Figure 2.3 The embryonic process in nature

The first condition so that the complete process keeps going is that there must be replicators, i.e. entities capable of self-replication, like DNA molecules. The second condition is that there must be an embryonic process. The developed organism or phenotype, and the replicators must be able to wield some phenotypic power over their world, such that some of them are more successful at replicating themselves than others.

It is important to understand that genes, the basic constituents of the genome, act on two quite different levels: they participate in the embryonic process, influencing the development of the phenotype in a given generation, and they participate in genetics, having themselves copied down the generations (reproduction). This is epitomised by an empirical separation between the disciplines of genetics and embryology. Genetics is the study of the vertical arrows in figure 2.3, i.e. the relationship between genotypes in successive generations; while embryology is the study of the horizontal arrows, i.e. the relationship between genotype and phenotype in any one generation.

Research into self-reproducing machines, inspired by the ontogeny of living beings, began with von Neumann in the late 1940s. This line of research can be divided into five stages, placed along the ontogenetic axis.

1. Von Neumann [Neu66] and his successors Banks, Burks [Bur70], and Codd [Cod68] developed self-reproducing automata capable of universal computation (i.e., able to simulate a universal Turing machine) and of universal construction (i.e., able to construct any automaton described by an artificial genome). Unfortunately, the complexity of these automata is such that no physical implementation has yet been possible, and only partial simulations have been carried out to date [Sig89, Pas95].

2. Langton [Lan84] and his successors Byl [Byl89], Reggia et al. [Reg93], and Morita et al. [Mor97] developed self-reproducing automata which are much simpler and which have been simulated in their entirety. These machines, however, lack any computing and constructing capabilities, their sole functionality being that of self-reproduction.

3. Tempesti [Tem95] and Perrier et al. [Per96] developed self-reproducing automata inspired by Langton's work, yet endowed with finite or universal computational capabilities.

4. Mange et al. [Man96a] and Marchal et al. [Mar96] proposed a new architecture called **embryonics**, or embryonic electronics. Based on the three features usually associated with the ontogenetic process in living organisms (multicellular organisation, cellular division and cellular differentiation), they introduced a new cellular automaton, complex

enough for universal computation, yet simple enough for a physical implementation through the use of commercially available digital circuits. In addition to self-reproduction, this multicellular "organism" also exhibits self-repair capabilities, another bio-inspired phenomenon.

5. All the above machines are characterised by an asexual reproductive process; the genome is therefore haploid. Hikage et al. have discussed the use of a diploid genome [Hik97]. This idea, coupled with the recombination of genetic material from two parents, could be introduced within the embryonics framework, representing an ultimate phase with respect to reproducing machines.

2.3.3 Epigenesis

The ontogenetic program (genome) is limited in the amount of information that can be stored, thereby rendering the complete specification of the organism impossible. A well-known example is that of the human brain with some 10^{10} neurones and 10^{14} connections, far too large a number to be completely specified in the four-character genome of length 3×10^9. Therefore, upon reaching a certain level of complexity, there must emerge a different process that permits the individual organism to integrate the vast quantity of interactions with the outside world. This process is known as epigenesis, and primarily includes the nervous system, the immune system and the endocrine system. These systems are characterised by the possession of a basic structure that is entirely defined by the genome (the innate part), which is then subjected to modification through interactions of the individual with the environment (the acquired part). The epigenetic process can be loosely grouped under the heading of learning systems.

The nervous and immune systems have already served as inspiration for engineers. The nervous system has received the most attention, giving rise to the field of artificial neural networks. The immune system has inspired systems for detecting software errors [Xan95], as well as immune systems for computers [Kep94]. Immunity of living organisms is a major domain of biology; it has been demonstrated that the immune system is capable of learning, recognising, and, above all, eliminating foreign bodies that continuously invade the organism. This feature leads us to surmise that the immune system, if implemented as an engineering model, can provide a new tool suitable for confronting dynamic problems, involving unknown, possibly hostile, environments. Tyrrell proposes the term ***immunotronics*** to name those electronic systems capable of self-diagnosis and self-healing by applying mechanisms equivalent to those found in the immune system [Tyr99].

2. 4 Evolvable Hardware

A careful examination to the work carried out to date under the heading 'evolvable hardware' makes evident that this has mostly involved the application of evolutionary algorithms to the synthesis of digital systems [San96a]. From this point of view, evolvable hardware is a sub-domain of artificial evolution, where the final goal is the synthesis of an electronic circuit.

Taken as a design methodology, evolvable hardware offers a major advantage over classical methods; the designer's job is reduced to that of specifying the circuit requirements and the basic elements, whereupon evolution "takes over" to "design" the circuit. Currently, most evolved digital designs are sub-optimal with respect to traditional methodologies, however, improved results are continuously attained. By examining the work carried out to date, it is possible to derive a rough classification of current evolvable hardware, in accordance with the genome encoding, i.e. the circuit description, and the calculation of a circuit's fitness.

2.4.1 Classification of evolvable hardware by genome encoding

- **High-level languages.** The first works used a high-level functional language to encode the circuits in question, a representation far-removed from the structural (schematic) description. The work presented in [Hem96] uses a high-level hardware description language (HDL) to represent the genomes. In [Kit96] Kitano used the rewriting operation, in addition to crossover and mutation, to enable the formation of a hierarchical structure.

- **Low-level languages.** The idea of directly incorporating within the genome the bit string representing the configuration of a programmable circuit was expressed early on by [Gar96], though without demonstrating its actual implementation. As a first step one must choose the basic logic gates (e.g., AND, OR, NOT), and suitable codify them, along with the interconnections between gates, to produce the genome encoding. An example of this approach is the work presented by Thompson in [Tho96a]. Higuchi et al. used a low-level bit string representation of the system's logic schema to describe small-scale PALs, where the circuit is restricted to a logic sum of products [Hig96]. The limitations of PAL circuits have been overcome to a large extent by the introduction of FPGAs, as used by Thompson in [Tho96b].

The use of a low-level circuit description that requires no further transformation is an important step forward since this potentially enables placing the genome directly in the

actual circuit, thus paving the way toward truly evolvable hardware. However, up until recently, FPGAs had introduced their own share of problems:

- The genome's length was on the order of tens of thousands of bits, rendering evolution practically impossible using current technology.
- One still had to extend the genome into a logic schema, a phase for which automatic methods do not exist.
- Within the circuit "space", consisting of all representable circuits, a large number were invalid, e.g. containing short circuits.

With the introduction of new families of FPGAs, like Xilinx 6200, these problems have been attenuated [Tho96b]. As with previous FPGAs families, there is a direct correspondence between the bit string of a cell and the actual logic circuit, however, this now always leads to a viable system. Moreover, as opposed to previous FPGAs where one had to configure the entire system, the new families permit the separate configuration of each cell, a markedly faster and more flexible process. Thompson has employed this latter characteristic to reduce the genome's size while introducing real-time, partial system reconfigurations [Tho96b]. Unfortunately Xilinx has withdrawn the 6200 family from the market. Its successor, the Virtex family, also offers partial reconfiguration characteristics, but its architecture does not satisfy the necessities of the evolvable hardware community, as well as the 6200 family. Details about FPGAs' architecture will be given in the next chapter.

2.4.2 Classification of evolvable hardware by fitness calculation

- **Offline evolvable hardware.** The use of a high-level language to represent the genome implies some transformation of the encoded system in order to evaluate its fitness. Fitness evaluation is carried out by simulation, with only the final solution found by evolution actually implemented in hardware. This form of simulated evolution is known as offline evolvable hardware [San96b].
- **Online evolvable hardware.** As noted above, the low-level genome representation enables a direct configuration (and reconfiguration) of the circuit, thus entailing the possibility of using real hardware during the evolutionary process. This approach has been referred to as online evolution in the works presented in [San96a].

Examining work carried out to date it is possible to identify a number of common characteristics that span both online and offline systems, which often differ from biological evolution:

- Evolution pursues a predefined goal: the design of an electronic circuit, subject to precise specifications; upon finding the desired circuit, the evolutionary process terminates.
- The population has no material existence; at best, in online evolution, there is one circuit available, onto which individuals from the (offline) population are loaded one at a time, in order to evaluate their fitness.
- The absence of a real population in which individuals coexist simultaneously entails notable difficulties in the realisation of interactions between "organisms". This results in a completely local fitness calculation, whereas nature exhibits a co-evolutionary scenario.
- In solving a well-defined problem, like the search for a specific combinatorial or sequential logic system, there are no intermediate approximations. Fitness evaluation is achieved by consulting a lookup table that contains the complete description of the circuit in question. This casts some doubts into the utility of using an evolutionary process, since one can directly implement the lookup table in a memory device, a solution which may often be faster and cheaper.
- The evolutionary mechanisms are carried out outside the resulting circuit. This includes the genetic operators (selection, crossover and mutation) as well as fitness calculation. As for the latter, while online evolution uses a real circuit for fitness evaluation, the fitness values themselves are stored elsewhere.
- The different phases of evolution are carried out sequentially, controlled by a central software unit.

These differences demonstrate that, although inspired by nature, bio-inspired systems do not have to strictly adhere to nature's solutions. As an example, consider the issue of Lamarckian evolution, which involves the direct inheritance of acquired characteristics. While the biological theory of evolution has shifted from Lamarckism to Darwinism, this does not preclude the use of artificial Lamarckian evolution. Thus, "deviations" from what is strictly natural may definitely be of use in bio-inspired systems.

In the future, radical new technologies, like nanotechnology [Dre90], will allow the physical implementation of 3-D microscopic machines able to perform adiabatic computation (computation without heat generation), at speeds beyond the limits of today's microelectronics [Gar96]. Such capabilities combined with the knowledge that today is being generated through simulation, might be the crucial missing elements needed to achieve the so longed goal of creating an artificial living being.

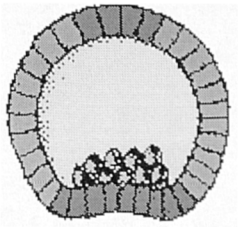
Artificial organisms will have fault tolerance as an inherent characteristic, therefore any effort directed towards the creation of artificial life will, indirectly, contribute to the development of highly reliable systems. Once artificial organisms become a reality, our world will never be the same. Certainly, it will be much better [Kel94].

2. 5 Summary

Fault tolerance is a common attribute of biological organisms; therefore drawing inspiration from nature is a promising alternative for the design of human-made fault-tolerant systems. The study of bio-inspired systems was practically born with science itself, but the lack of appropriate technologies made impossible the implementation of the ultimate bio-inspired system: an artificial organism. Modern technologies allow more serious investigations in this area; an effort consolidated under the name of Artificial Life of ALife.

The POE model classifies bio-inspired systems according to their level of organisation. Phylogeny (P) deals with the evolution of populations, Ontogeny (O) deals with the development of multicellular individuals, and Epigenesis (E) deals with learning mechanisms that help individuals to adapt to the environment. All the work done in ALife can be classified in one or several of these categories. It is expected that an artificial organism will include sub-systems from all the divisions in the POE model.

Bio-inspired electronic systems are being developed under the name of Evolvable Hardware or EHW. These systems are evolved rather than designed in the traditional way. If a system evolves continuously, then it can autonomously adapt to changes in the environment, for example variations in temperature or faults in one or some of its components. This adaptability turns EHW into a promising approach to the design of bio-inspired fault-tolerant electronic systems. The following chapters present the implementation of a system governed by these principles.



CHAPTER 3

EMBRYONICS: A CONFLUENCE OF IDEAS

Embryonics is a proposal for a bio-inspired cellular architecture with inherent fault tolerance properties. This chapter gives a brief introduction to the fundamental ideas that give shape to the embryonics architecture. The biological inspiration comes from the development of embryos and the central dogma of molecular biology. The technological resources employed to implement embryonics are cellular architectures, field programmable gate arrays and ordered binary decision diagrams. Each one of these subjects is treated in a separate section.

3.1 Introduction

In previous chapters it has been established that nature can inspire new ways of achieving fault tolerance in electronic systems. This chapter introduces **Embryonics**, a bio-inspired reconfigurable cellular architecture that offers a simple yet effective solution to the problem of incorporating fault tolerance in electronic digital systems.

The hypothesis of this work is that embryonic arrays can be considered an attractive alternative for improving the fault tolerance of cellular architectures [Ort97b, Ort98b].

In the context of the POE model, embryonics belongs to the ontogeny class because it is a family of fault-tolerant field programmable processor arrays (FPPAs) inspired by the mechanisms involved during the development of embryos [Man96a]. By adopting certain features of cellular organisation, and by transposing them to the two-dimensional world of integrated circuits on silicon, embryonics shows that properties unique to the living world,

such as self-reproduction and self-repair can also be applied to integrated circuits. Hugo de Garis coined the word embryonics as an acronym for embryological electronics [Gar93].

Any bio-inspired system must have a biological as well as a technological support. Embryonics is not the exception. A multitude of disciplines and techniques converge to give shape to the embryonics project. On the biological side, both embryology and the central dogma of molecular biology inspired the main features of embryonic arrays. These ideas are supported by a technological backbone that includes 2-D cellular systems and field programmable gate array design. Both elements are combined in the embryonics architecture. Finally, embryonics arrays are proposed to solve a particular problem namely, the hardware implementation of ordered binary decision diagrams. Figure 3.1 shows the convergence of ideas giving rise to embryonics and the particular problem that embryonics can solve. Numbers in parenthesis indicate the sections in this chapter that introduce the corresponding subject.

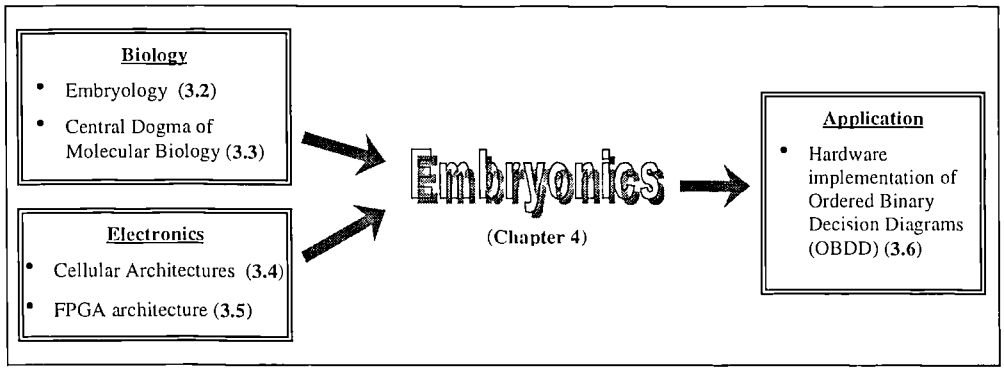


Figure 3.1 Embryonics: A confluence of ideas

The following sections expose the concepts and ideas involved in the embryonics project. They are independent from each other; therefore it is possible to read them in any order.

Embryo development and the central dogma of molecular biology are presented in sections 3.2 and 3.3 respectively. These sections do not cover any of the subjects exhaustively, only the central ideas and mechanisms involved are discussed. However, the references given cover the subjects thoroughly. Sections 3.4 and 3.5 introduce the technological background needed to understand the physical implementation of embryonic arrays. Cellular arrays architectures, including cellular automata, systolic arrays and wavefront arrays are treated in section 3.4. Modern microelectronics allows the integration of these arrays in one chip and, if programmability capabilities are added to these components, the result is a **field programmable processor array** (FPPA). The precursors of FPPAs are the **field programmable gate arrays** (FPGA), i.e. programmable arrays where the basic elements perform simple logic functions. Section 3.5 presents current FPGA architectures as the point

of departure for the implementation of embryonic arrays. A practical application of embryonic arrays is the physical implementation of **Ordered Binary Decision Diagrams** (OBDD). They are covered in section 3.6.

3.2 Embryo development

The embryonic development of multicellular organisms is one of evolution's greatest achievements. The process by which organisms as complex as human beings are constructed from a single cell amazes even the most specialised embryologist. The ultimate details of how this complex, yet reliable mechanism is carried out are still a mystery; but the overall picture has been revealed over the years by dedicated and imaginative researchers like Wolpert and Nüsslein-Volhard, to mention just two.

Cells are the basic units of life. Animals are made up of specialised cells, such as blood cells, cartilage cells, fat cells, muscle cells, nerve cells. Humans have about 350 different cell types. All the cells in an organism are created by the successive division of a single cell, the fertilised egg [Wol91].

The embryo's development starts with cell divisions that cleave the egg and result in a hollow spherical structure: **the blastula**. The blastula is then moulded by cellular activities into all the shapes that emerge during development. The blastula gives no visible indication of the organism it will develop into. It is only after the next stage, **gastrulation**, that the form of the organism begins to emerge.

Gastrulation occurs in the development of all animals. It is the process that occurs when the cells of the blastula rearrange and move so that the simple and often spherical or flat embryo is transformed into something approaching the form from which the animal will develop. It is only after gastrulation that the organs, like limbs, liver, and eyes, begin to develop. Contractions, changes in adhesion, cell movement, and growth, are the cellular activities that go to mould the form of the embryo. These cell activities are used again and again, and what makes organs different is how these activities are organised in space and time. That is the problem of pattern formation.

All vertebrates have basically the same building blocks but they are put together in different ways. The principle of different spatial patterning accounting for the differences in animals applies right across the vertebrates. There are some differences in the cell types that make up fish, frogs, birds, and humans; nevertheless, the main difference lies in the spatial organisation of the cells.

Cells differentiate in the early embryo according to their relative position with respect to the other cells. By means of chemical gradients cells can acquire positional information within the embryo. In addition, each cell has a set of instructions, analogous to genetic information, which lists what every cell must do in every position. The cells just look up their position in this set of instructions and behave accordingly. Following these simple mechanisms it is possible to generate any pattern that is required, from faces to limbs.

The "co-ordinate system" just described allows another remarkable property of the early embryo to emerge: the process known as **regulation**. Regulation is the ability of the embryo to develop normally even when some portions are removed or rearranged. In general, if cells of vertebrate embryos are moved from one part to another of the early embryo they develop according to their new location and not from where they were taken. Their fate is dependent on their new position in the embryo: they respond to their new set of co-ordinates. For example, in the mouse egg and at least up to the 16-cell stage all the cells seem equivalent with no fixed fate. It is possible to rearrange the cells of the early mouse embryo in numerous combinations and normal development will still occur. In humans, identical twins rarely arise from the separation into two cells at the two-cell stage. Instead, the separation occurs much later when the embryo is made up already of many hundreds of cells. This means that in human embryos even when there are several hundred cells present the fate of the cells is not fixed and if divided into two, two normal embryos can still develop [Wol91].

The fate of the cells becomes, with time, more and more restricted until it is effectively fixed. The cells acquire an autonomous developmental programme and no longer respond to new positional cues. The process by which cells have their fate fixed is known as **determination**. Determination involves subtle chemical changes that turn on and off genes, making cells different.

Once the cells in an embryo have been differentiated messages are spread by timed releases of chemicals that tell a cell which type of cell it should be. Each cell has a look up library, the code of DNA, to control its actions. A cell also has a chemical plant and chemical responses that act as its input and output devices. Different doses of chemicals and in different combinations cause a cell to act in different ways. Sometime small changes in chemical density can lead to radical changes in cell formation. This is, for example, how cell barriers are formed. It is by this mixture of co-operation and competition that complex structures, like a human baby, can be built up [Ste97].

3.3 The Central Dogma of Molecular Biology

A human being consists of approximately 60 trillion ($60x10^{12}$) cells. At each instant, in each of these 60 trillion cells, the genome, a ribbon of 2 billion characters, is decoded to produce the proteins needed for the survival of the organism. This genome contains the ensemble of the genetic inheritance of the individual and, at the same time, the instructions for both the construction and the operation of the organism. The parallel execution of 60 trillion genomes in as many cells occurs ceaselessly from the conception to the death of the individual. Faults are rare and, in the majority of cases, successfully detected and repaired [Man96a].

This process is remarkable for its complexity and its precision. Moreover, it relies on completely discrete processes: the chemical structure of DNA, the chemical substrate of the genome. DNA is a sequence of four bases (nucleotides) usually designated with letters A (adenine), C (cytosine), G (guanine), and T (thiamine). Each group of three bases (a codon) is decoded in the cell to produce a particular amino acid, a future constituent of the final protein.

As mentioned, the DNA encodes the ensemble of the genetic inheritance of the individual and, at the same time, the instructions for the construction and operation of the complete organism. In this sense DNA can be both information and physical medium.

DNA is a very long string-like molecule and it is packaged, with special proteins, in the form of chromosomes within the nucleus of the cell. Humans have 46 chromosomes, 23 from the father and 23 from the mother, each of which can be matched with its partner from the other parent. Each chromosome contains just one DNA molecule so there are exactly 46 molecules of DNA in the fertilised egg and all normal body cells.

In any living being every one of its constituent cells performs the same basic operation regardless of the particular function it is involved with; namely, each cell interprets the DNA strand allocated in its nucleus to produce the proteins needed for the survival of the organism. Proteins are particular sequences of amino acids; such sequences are stored in the DNA as successions of nucleotide triplets (codons). The DNA contains not only the instructions for making all the proteins but is also involved in the controlling of which protein should be made when and where.

Protein synthesis implies two mechanisms: transcription and translation of the DNA. During transcription, the sequence stored in the DNA is copied by the enzyme RNA polymerase into messenger RNA (mRNA). During translation, mRNA is bound to ribosomes inside the cell where transfer RNA (tRNA) carrying amino acids are attached to the mRNA. The ribosome

catalyses the bond between amino acids to build a molecule of the corresponding protein. When a cell reproduces, the offspring get a copy of its mother's DNA so that the complete process can be ceaselessly repeated. The flow of information from DNA to protein and from DNA of the parent to DNA of the offspring is known as the central dogma [Mur89]. Figures 3.2 and 3.3 show the processes of DNA's transcription and translation.

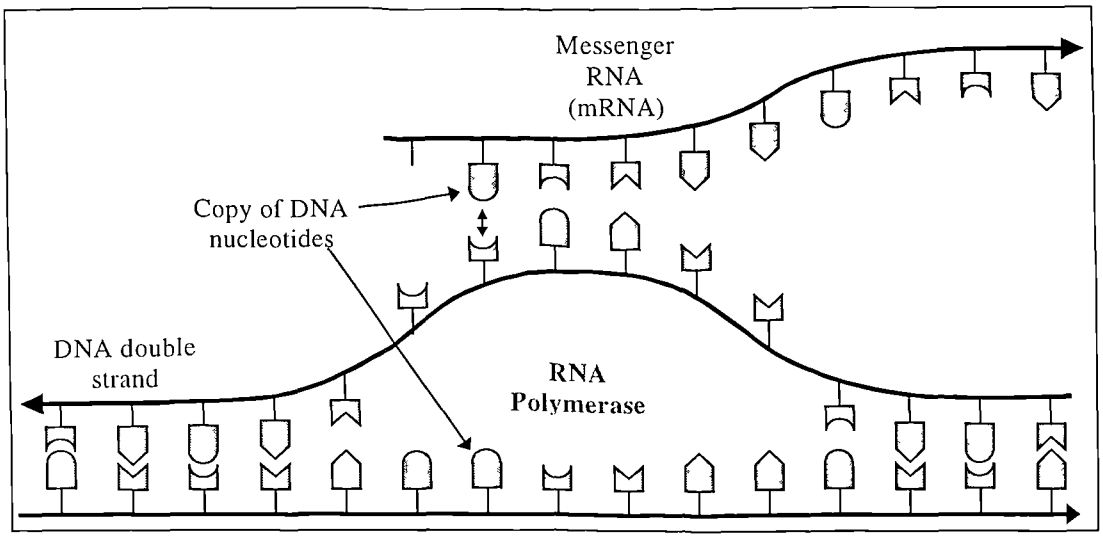


Figure 3.2 Transcription of DNA

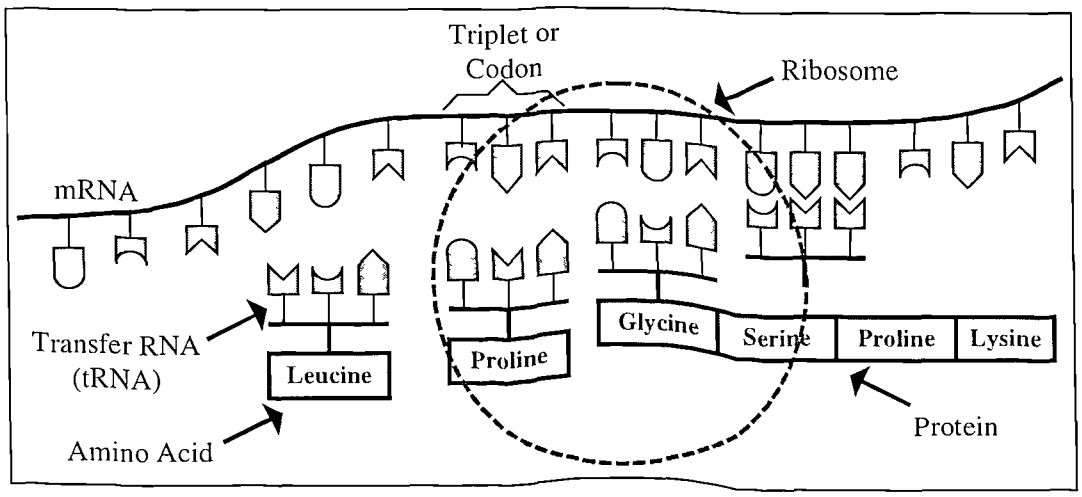


Figure 3.3 Translation of DNA into proteins

Although the DNA is identical in all the cells, only part of the strand is interpreted depending on the cell's function: red blood cells produce haemoglobin while liver cells produce albumin. Differentiation of cells will depend on the physical location of the cell with respect to its neighbours in the early embryo [Nus96, Wol91]. Control of protein synthesis is the central issue in cell differentiation and development.

Control of protein synthesis can occur at several different points in the sequence of steps that leads from the DNA code to a fully formed protein. The first step, transcription, is considered the most important. Control of transcription is done in two main ways. Proteins can bind to sites on the DNA at the beginning of the gene, known as the **promoter**, and so initiate transcription. On the other hand, there are proteins that bind near the promoter preventing its transcription. The control of transcription (turning on and off of genes), and so controlling the synthesis of specific proteins, involves signal molecules that enter the cell nucleus from the cytoplasm.

Figure 3.4 represents the way DNA's information is organised. Arrows indicate the direction in which complexity increases, e.g. a set of nucleotides forms a codon.

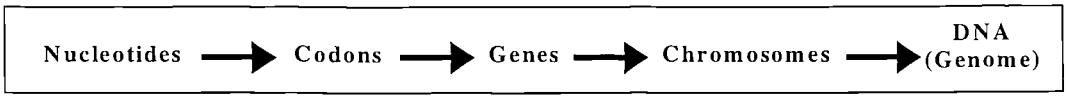


Figure 3.4 Structure of DNA's information

The aim of Embryonics is to transport this basic structure to the 2-dimensional world of cellular arrays using specifically designed FPGAs as building blocks. Figure 3.5 shows an equivalent representation of the architecture of field-programmable processor arrays.

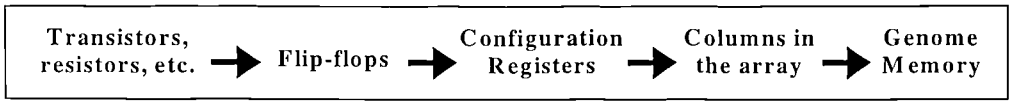


Figure 3.5 Structure of a field-programmable processor array

3.4 Cellular Architectures

It appears that the basic laws of physics relevant to everyday phenomena are now known. Yet there are many everyday natural systems whose complex structure and behaviour have so far defied even qualitative analysis. For example, the laws that govern the freezing of water and the conduction of heat have long been known, but analysing their consequences for the intricate patterns of snowflake growth has not yet been possible. Many complex systems can be broken down into identical components, each obeying simple laws, and the interaction of components that makes up the whole system gives forth very complex behaviours. In some cases these complex behaviours may be simulated in software with just a few components; but in most cases the simulation requires too many components, and this approach becomes impractical.

Another alternative is to distil the mathematical essence of the processes that generate complex behaviour. The hope in such an approach is to identify fundamental mathematical

mechanisms that are common to many different natural systems. Such commonality would correspond to universal features in the behaviour of very different complex natural systems. However, in order to discover the mathematical basis for the generation of complexity, one must identify simple mathematical systems that capture the essence of the process, a task that has been proven successful in very few cases. Chaos theory and non-linear systems analysis are common tools utilised in this approach [Kau96].

A third approach is the implementation of cellular systems in hardware. The throughput of these systems can easily be at least one order of magnitude better than that of software simulation. Cellular automata, systolic arrays and wavefront arrays are architectures that have been proposed for the implementation of hardware cellular systems.

3.4.1 Cellular automata

Cellular automata are discrete dynamical systems. The meaning of discrete is that space, time and the automaton's properties can have only a finite, countable number of states. The basic idea is not to describe a complex system from "above" using difficult equations, but simulating this system by interaction of cells following easy rules. In other words: Not to describe a complex system with complex equations, but let the complexity emerge by interaction of simple individuals following simple rules [Wol83]. Typical digital computers process data serially, cellular automata process a large number of bits in parallel.

The name von Neumann is now strongly associated with the old-fashioned, single-CPU computer architecture. Nevertheless, John von Neumann was also the major pioneer in parallel computing and self-reproducing artificial organisms via his research on arrays of computers or cellular automata (CA).

In 1944, von Neumann was introduced to electronic computing via a description of the ENIAC. Shortly after, he formed a group of scientists to work on problems in computers, communications, control, time-series analysis, and the communication and control aspects of the nervous system. In 1946 this group designed the EDVAC, which was the first design of a stored-program machine [Neu45].

By 1947, under the influence of the ideas on automata developed by Post and Turing, von Neumann had commenced his studies on the complexity required for a device or system to be self-reproductive. These studies also included work on the problem of designing a reliable system from unreliable parts; a field of study known today as "fault tolerant computing". At first, von Neumann investigated a continuous model of a self-reproducing automaton based on a system of non-linear partial differential equations. He also pursued the idea of a

kinematic automaton, which could, using a description of itself, proceed to mechanically assemble a duplicate from available pieces [Pre84].

At about the same time, mathematician Stanislaw Ulman was researching into pattern games for computers [Bru97]. Given certain fixed rules, the computer would print out ever-changing patterns. Many patterns grew almost as if they were alive. A simple square would evolve into a delicate coral-like growth. He called his patterns *recursively defined geometric objects*. Ulman's games were cellular games. Each pattern was composed of square (or triangular, or hexagonal) cells and the games were played on limitless chessboards. All growth and change of patterns took place in discrete jumps. From moment to moment, the fate of a given cell depended only on the states of its neighbouring cells.

Ulman suggested to von Neumann to construct an abstract universe for his analysis of machine reproduction. It would be an imaginary world with self-consistent rules, as in Ulman's games. It would be a world complex enough to embrace all the essentials of machine operation, but otherwise as simple as possible. Von Neumann adopted an infinite chessboard as his universe. Each square cell could be in any of a number of states corresponding roughly to machine components. A "machine" was a pattern of such cells.

The first cellular automaton was conceived by von Neumann in the late forties. By 1952 he had put his ideas in writing and in 1953 described them more fully in his Vanuxem lectures at Princeton University. Unfortunately, his premature death in 1957 prevented him from completely achieving his goals. Nevertheless, the details of von Neumann's cellular construction were completed and published after his death by A.W. Burks [Bur70], who worked with von Neumann on the logical design of EDVAC.

Von Neumann's original construct for a self-reproducing cellular automaton required that each computer in the array support a set of 29 states. The array itself required some 200,000 computers performing functions such as: tape reading arms, "pulsers", clocks, encoders and decoders. This degree of complexity was needed since von Neumann wanted to design his automaton as a universal computing system or Turing machine, i.e. a construct capable of performing any desired calculation [Pre84]. Of course, the more complex the machine which is to accomplish the construction, the more complex the algorithm for building that machine will be, and, therefore, the longer the tape which contains the description of the machine. Thus, there was a genuine incentive for finding "simple" machines that are nonetheless still capable of self-reproduction.

For his doctoral research at the University of Michigan, E.F. Codd [Cod68] set out to reduce the complexity of von Neumann's machine. He was able to design a construction universal

configuration that requires just 8 states per cell. Although simpler than von Neumann's, Codd's machine is still as complex as a modern digital computer, and as far as is known, neither Codd's nor von Neumann's machines have actually been run under real simulation on a computer, only partial simulations have been reported [Sig89, Pas95, Hae97].

The next significant event in the history of self-reproducing automata was the development of the automaton commonly referred to as "Langton's loop" [Lan84]. By dropping the requirements of computational and construction universality, Langton created an automaton capable of non-trivial self-replication, i.e. an automaton where the replication is actively directed by the automaton itself, rather than being a mere consequence of the transition rules. Langton's research was followed by a series of works attempting either to further simplify Langton's loop [Reg93] or to modify it in such a way that it would be capable of performing some useful work, beyond that of self-reproduction [Tem95].

All the work on self-reproducing cellular automata share a characteristic that must be found in any self-reproducing system (and is certainly found in molecular self-reproduction): the configuration treats its stored information in two different manners: interpreted, as instructions to be executed (*translation*), and uninterpreted, as data to be copied (*transcription*). In nature, each biological cell keeps the information of how to construct the being of which it belongs to in the DNA strand allocated in its nucleus. When reproduction takes place RNA copies the DNA from the mother cell to the descendants; this is the transcription process. Once copied, the ribosomes interpret each part of the DNA in order to build the proteins needed for the development of the complete organism; this is the translation process [Zif83].

Cellular automata have five fundamental defining characteristics:

1. They consist of a discrete lattice of cells.
2. They evolve in discrete time steps.
3. Each cell takes on a finite set of possible values.
4. The value of each cell evolves according to the same deterministic rules.
5. The rules for cell evolution depend only on a local neighbourhood of cells around it.

With these characteristics, cellular automata provide rather general discrete models for homogeneous systems with local interactions. They may be considered as idealisations of partial differential equations, in which time and space are assumed discrete, and dependant

variables taken on a finite set of possible values. These considerations allow a mathematical treatment of systems based in cellular automata.

Different definitions of neighbourhood for a cell are possible; all of them consider the cell itself as part of its neighbourhood. Considering a two dimensional lattice the following definitions are commonly accepted:

- **Von Neumann neighbourhood**. Only the cells on the cardinal points (North-South-East-West) are considered therefore, the number of cells involved to calculate the next state of any cell is 5. The radius of this definition is 1, since only the next layer is considered. (Fig.3.6a)
- **Moore neighbourhood**. The Moore neighbourhood is an enlargement of the von Neumann neighbourhood containing the diagonal cells too. In this case the number of cells considered for calculating the next state is 9, and the radius is 1. (Fig.3.6b)
- **Extended Moore neighbourhood**. In this instance the radius can take any value greater or equal to 2, therefore, the neighbourhood reaches over the distance of the next adjacent cells. (Fig.3.6c)
- **Margolus neighbourhood**. A completely different approach: considers 2x2 cells of a lattice at once. In this case the rules for assigning the next state become more complex and difficult to express in mathematical language [Mar84].

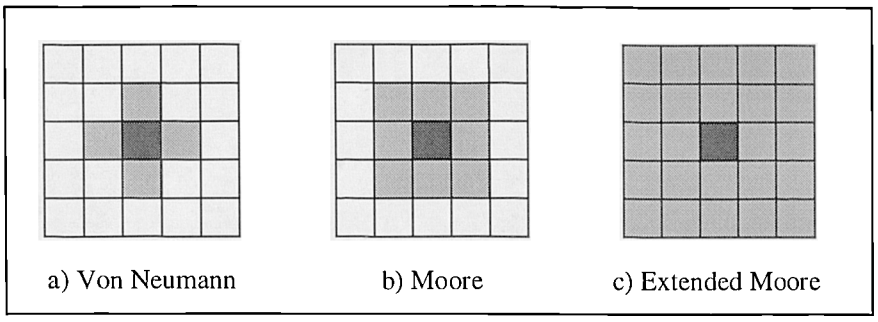


Figure 3.6 Different types of neighbourhood for cellular automata

Cellular Automata Applications

Besides the aforementioned application of designing self-reproducing systems, some research has been done to exploit the properties of cellular automata. Here are some of the most representative works.

Game of Life

The game of Life was one of the first applications showing that cellular automata are capable of producing dynamic patterns and structures. Life is played on a two dimensional lattice

with binary cell states, Moore neighbourhood and arbitrary border conditions. A cell in state 1 is said to be alive, while a cell in state 0 is dead. J. Conway introduced the following rules:

1. A cell that is dead at time step t, becomes alive at time $t+1$ if exactly three of the eight neighbouring cells were alive at time t.
2. A cell that is alive at time t dies at time $t+1$ if at time t less than two or more than three cells are alive, i.e. the cell dies of either isolation or overcrowding.

Though these rules are rather simple, a vivid set of occurring patterns can be observed. Some patterns flicker infinitely between two states, like blinkers; some are static blocks, snakes and ships; others move over the lattice and vanish into the infinity of the lattice.

Ising Model

A different application is the CA-ising model that can be used to simulate ferro-magnetism. Every cell stands for the spin of a small magnet, where the state 1 may represent an "up" vector and the state 0 the "down" vector. The orientation of the spin is variable and depends on the local neighbourhood. Temperature plays an important role in this model; two conditions can be named:

- If temperature T > Curie-temperature, then the second law of thermodynamics is dominating creating disorder (chaos).
- If T < Curie-temperature, then the force between spins is dominating and spins tend to build order.

CAs can be used to simulate this system with the additional difficulty that the spin (energy) of the whole system has to remain constant [Tof87].

Billiard and gas models

The dynamic of cellular automata can be used to simulate the behaviour of particles (gas molecules or billiard balls). The construction of a gas model is similar to the so-called billiard automaton. These kinds of systems use a Margolus neighbourhood to simulate the process. The rules are based on 2x2 parts of the lattice. A selection of rules is shown in figure 3.7.

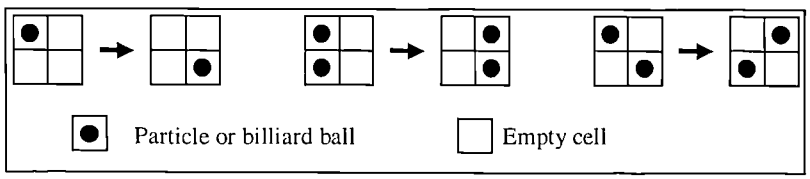


Figure 3.7 Some rules for a Margolus neighbourhood

3.4.2 Systolic and Wavefront Arrays

Modern circuit manufacturing techniques allow the construction of million-transistor chips and with this the integration of highly parallel computing structures has become viable. Such computing systems have structural properties that are suitable for VLSI implementation. Parallel structures imply a basic computational element repeated perhaps hundreds or thousands of times. This architectural style immediately reduces the design problem by similar orders of magnitude because of its simplicity and regularity.

The choice of an appropriate architecture for any electronic system is very closely related to the implementation technology. This is especially true in VLSI. The constraints of power dissipation, I/O pin count, relatively long communication delays, difficulty in design and layout are much more critical in VLSI than in other technologies. However, VLSI offers very fast and inexpensive computational elements with some unique and exciting properties [Hwa85].

Properly designed parallel structures that need to communicate only with their nearest neighbours gain the most from VLSI. Precious time is lost when modules that are far apart must communicate. For example, the delay in crossing a chip on polysilicon, one of the three primary interconnect layers on an NMOS chip, can be 10 to 50 times the delay of an individual gate. Two architectures that are particularly well suited to be implemented in silicon are *systolic arrays* and *wavefront arrays*.

Systolic arrays

Systolic arrays belong to the generation of VLSI/WSI (Very Large-Scale Integration/Wafer Scale Integration) architectures for which regularity and modularity are important to area-efficient layouts. Kung and associates at Carnegie-Mellon University developed the systolic architectural concept [Kng82]. Since its introduction many versions of systolic processors have been designed by universities and industrial organisations.

A systolic system consists of a set of interconnected cells, each capable of performing some simple operation. Cells in a systolic system are typically interconnected to form a systolic array or a systolic tree using simple, regular communication and control structures. These simple interconnection and control schemes have substantial advantages over more complex designs and implementations. Information in a systolic system flows between cells in a pipeline fashion, and communication with the outside world occurs only at the boundary cells.

Systolic arrays took their name from an analogy with the human circulatory system where the heart sends and receives a large amount of blood through the veins and arteries. The phase during which the heart contracts itself to pump the blood through the arteries is called the systole phase. In this context the heart can be viewed as a source and destination of data, a sort of global memory, and the network of arteries and veins as an array of processors and links. Similarly, systolic algorithms schedule computations in such a way that a data item is not only used when it is input but also is reused as it moves through the pipelines in the array. In systolic arrays pipelined computations take place along all dimensions of the array and result in very high computational throughput. As a consequence, the processing and input/output bandwidths can be balanced, especially in compute-bound problems that have more computations to be performed than they have inputs and outputs [For87].

Figure 3.8 shows the basic principle of a systolic array [Hwa85].

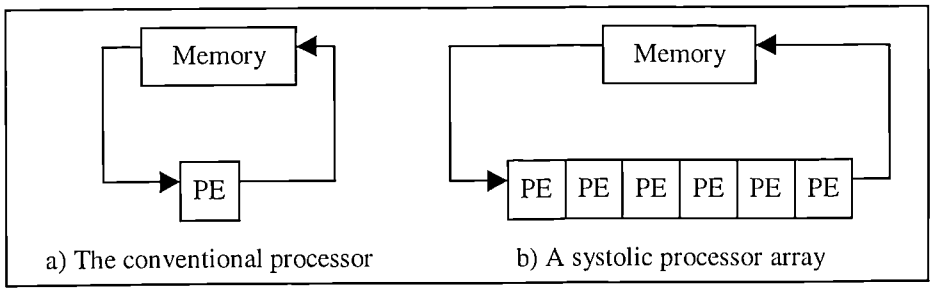


Figure 3.8 The systolic array principle

Figure 3.8 shows how by replacing a single processing element (PE) with an array of processing elements (e.g. linear, 2-D and 3-D), a higher computation throughput can be achieved without increasing memory bandwidth. The function of the memory in the diagram is analogous to that of the heart: it "pulses" data through the array of PEs. The essence of this approach is to ensure that once a data item is brought out from the memory it can be used effectively at each cell it passes. This is possible for a wide class of compute-bound computations where multiple operations are performed on each data item in a repetitive manner [Hwa85].

In addition to data pipelining, systolic arrays are also characterised by *computational pipelining*, in which information flows from one PE to another in a pre-specified order. This information can be interpreted by the receiver as data, control, or a combination of both. Each output is computed by the execution, at different times and in a pre-determined sequence, of several operations in a number of PEs. The execution is performed in such a way that the output generated by one PE is used as an input by a neighbouring one. While operations can occur as data flow through each processor, the overall computation is not a

dataflow computation, since the operations are executed according to a schedule determined by the systolic array design. Correct timing is a critical issue when designing algorithms for systolic arrays.

Wavefront arrays

The burden of synchronising an entire systolic computing network becomes heavy for very large arrays. A simple solution is to take advantage of the dataflow computing principle, which is natural to signal processing algorithms and which leads the designer to wavefront array processing. The wavefront array combines the systolic pipelining principle with the dataflow computing concept. The wavefront arrays can be viewed as a static dataflow array that supports the direct hardware implementation of regular dataflow graphs. Exploitation of the dataflow principle makes the extraction of parallelism and programming for wavefront arrays relatively simpler. Conceptually, the requirement for correct timing in the systolic array is now replaced by a requirement for correct sequencing in the wavefront array [Kun87].

There are two approaches to deriving wavefront array algorithms: one is to trace and pipeline the computational wavefronts; the other is based on a data flow graph (DFG) model. "Computational wavefront" means smooth data movement in a localised communication network. The computing network serves as a data-wave-propagating medium. A wavefront in a processor array corresponds to a mathematical recursion in an algorithm. Successive pipelining of wavefronts through the array will accomplish the computation of all recursions. The computational wavefronts are similar to electromagnetic wavefronts, since each processor acts as a secondary source and is responsible for the activation of the next front. This means that the computation is *data-driven*.

Note that the major difference between a wavefront array and a systolic array is the data-driven property. In a wavefront arrays there is no global timing reference, and yet the order of task sequencing is correctly followed. In the wavefront architecture the information transfer between a PE and its immediate neighbours is by mutual convenience. Whenever data is available, the transmitting PE informs the receiver, and the receiver accepts the data whenever required. It then communicates with the sender to acknowledge that the data have been consumed. This scheme can be implemented by means of a simple handshaking protocol, which ensures that the computational wavefronts propagate in an orderly manner instead of crashing into one another [Kun82]. Since there is no need to synchronise the entire array, a wavefront array is truly architecturally scalable. Another advantage of wavefront arrays is the low power consumption associated to its asynchronous mode of operation. In

asynchronous systems, power is only consumed where it is needed, whereas in synchronous designs the global clock continuously drives the switching logic [Hol82].

On the other hand, a wavefront array and a systolic array are identical in terms of regularity, modularity, local interconnection, and pipelinability. They both consist of modular processing units with regular and local interconnections. Their computing networks may be extended indefinitely. They exhibit a linear-rate speedup; i.e. they achieve a speedup, in terms of processing rates, proportional to M, where M is the number of PEs.

In summary, a simple way to relate the wavefront array to its systolic counterpart is:

Wavefront array = Systolic array + Dataflow computing [Kun87]

Systolic and Wavefront Arrays Applications

Both wavefront and systolic arrays share the important common feature of using a large number of modular and locally interconnected processors for massively pipelined and parallel processing. Table 3.1 presents a list of applications for which systolic and wavefront designs are available [For87].

Signal and Image Processing and Pattern Recognition	
• FIR, IIR filtering and 1-D convolution	• 2-D convolution and correlation
• Discrete Fourier Transform	• Interpolation
• 1-D and 2-D median filtering	• Geometric warping
• Feature extraction	• Order statistics
• Minimum-distance classification	• Covariance matrix computation
• Template matching	• Seismic signal classification
• Cluster analysis	• Syntactic pattern recognition
• Radar signal processing	• Curve detection
• Dynamic scene analysis	• Scene matching
Matrix Arithmetic	
• Matrix-matrix multiplication	• Matrix triangularisation
• QR decomposition	• Sparse-matrix operations
• Solution of triangular linear systems	
Non-Numeric Applications	
• Data structures: stacks and queues sorting	• Connected components
• Graph algorithms: Transitive closure, minimum spanning trees	• Language recognition
• Dynamic programming	• Arithmetic arrays
• Relational database operations	• Algebra

Table 3.1 Applications of systolic and wavefront arrays

3.4.3 Fault Tolerance in Cellular Systems

The parallel structures mentioned in the previous sections are good candidates to be implemented in silicon because of their regularity and the relative simplicity of the processing unit. In recent years the idea of parallel computers on a chip has become feasible thanks to the advances in VLSI and WSI technologies. Nevertheless, production yields of VLSI circuits are far from being optimum; therefore, reconfiguration techniques have been explored for the past few years in order to provide VLSI processor arrays with fault tolerance [Neg89, Lei85, Lom89].

All fault tolerance techniques for hardware systems rely on the use of spare components to substitute failing elements. In the past, the cost associated with this redundancy has prevented the widespread use of fault-tolerant hardware. However, in the case of VLSI processor arrays, redundancy comes for free because not all the cells available in the array are used on every application.

Fault tolerance in processor arrays implies the mapping of a logical array into a physical non-faulty array; i.e. every logical cell must have a correspondent physical cell [Gro94]. When faults arise, a mechanism must be provided for reconfiguring the physical array such that the logical array can still be represented by the remaining non-faulty cells. All reconfiguring mechanisms are based on one of two types of redundancy: Time redundancy or hardware redundancy [Che90b].

In time redundancy the tasks performed by faulty cells are distributed among its neighbours. In this scheme the application must allow graceful degradation in performance. When reconfiguration takes place, processors dedicate some time performing their own tasks and some performing faulty cells' functions. Nevertheless, the algorithm being executed must be flexible enough so as to allow a simple and flexible division of tasks.

In hardware redundancy physical spare cells and links are used to replace the faulty ones. Therefore, reconfiguring algorithms must optimise the use of spares. In the ideal case a processor array with N spares must be able to tolerate N faulty cells but, in practice, limitations on the interconnection capabilities of each cell prevents this goal from being achieved. Some prevalent strategies used to reconfigure cellular systems are Cell-elimination, Row/Column-elimination and Embryonics.

Most hardware redundancy reconfiguration techniques rely on complex algorithms to re-assign physical resources to the elements of the logical array. In most cases these algorithms are executed by a central processor, which also performs diagnosis functions and co-

ordinates the reconfiguration of the physical array [For85]. This approach has demonstrated to be effective, but its centralised nature makes it prone to collapse if the processor in charge of the fault tolerance functions fails.

An alternative approach is to distribute the diagnosis and reconfiguration mechanisms among all the cells in the array. In this way no central agent is necessary and the time response of the system improves. This mechanism resembles that found in biological cellular systems and will be explained in the next chapter.

3.5 Field-Programmable Gate Arrays

In recent years a new approach to hardware fault tolerance is being explored: The use of programmable logic to implement self-testable and self-reconfigurable circuits. To reach this point technology has mainly evolved in two areas: Design of high-density programmable circuits and improvement of CAD programs to assist the design and verification of the user's applications. Field Programmable Gate Arrays (FPGA) and the VHDL language are the corresponding prime technologies.

FPGAs resulted from the evolution of previous forms of programmable logic. PALs and PLDs were the dominant technologies in the 70s and 80s respectively. The design engineer was no longer constrained by the standard functions offered by standard TTL products. If a logic chip was not available for a specific function the designer could take a PAL device and create his own chip with the required logic function for a specific application. PAL's architecture consisted of an AND-OR array and some inverters. The inputs to the ANDs were the programmable bits.

Programmable Logic Devices (PLD) improved the architecture of PALs adding registers and feedback lines form the outputs to the AND-OR array [San96b]. Having these programmable chips, a programming machine and support software, the digital designer could create a customised logic system. The advantage to the user was a reduction of approximately 3 to 1 in the chip count of a finished design. Other advantages were the cover up for errors in printed circuit board layout, simpler PCB layout, and shorter time to market of final products [Jay93a]. There still remained a problem, though, the PLDs were limited in the amount of input and output buffers, on chip logic and registers.

Xilinx Inc. addressed the deficiencies in the PAL product by offering a static RAM based FPGA with a larger amount of input and output resources and on chip registers. Even with the early Xilinx devices six or seven PAL devices could be absorbed into one Xilinx chip. There are additional advantages in an SRAM based product, such as the capability of being reprogrammed to change the logic function in circuit. For example, a host processor could

configure an FPGA with a diagnostic function and reconfigure the device as a bus interface at a later time, thus removing the need for logic duplication.

3.5.1 FPGAs Architecture

From the architectural point of view, complex PLDs are based on EPROM technology. Consequently, the configuration data are non-volatile and erasable. The devices comprise a small number of coarse logic blocks, based on the PAL macrocell, and employ simple, fast interconnect that is relatively easy to route. In contrast, FPGAs usually have a large number of simple logic blocks that communicate via complex and fragmented interconnect that often detracts significantly from system performance. There are three architectural styles [Yor93] referred to as Red Square, Terraced and Manhattan, as shown in figure 3.9.

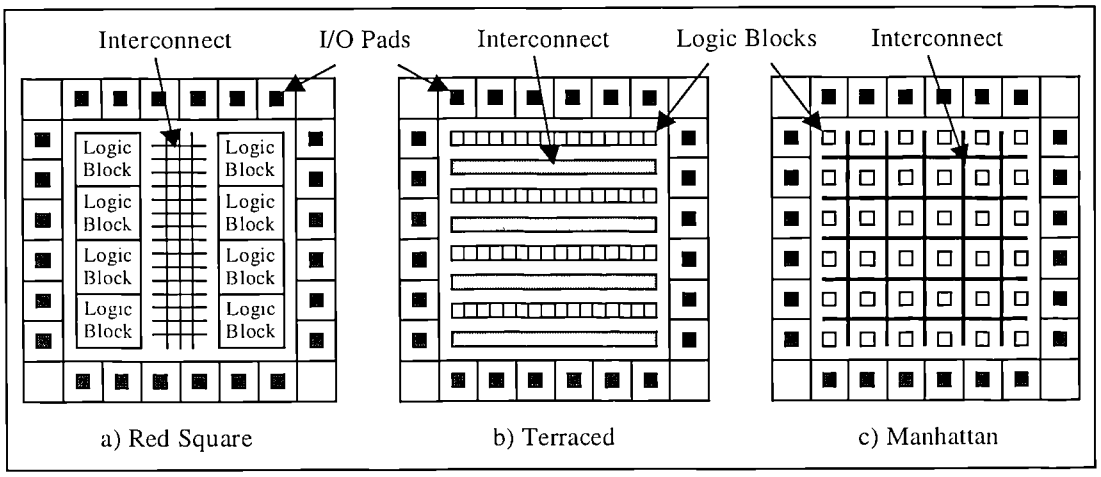


Figure 3.9 Architectural styles for programmable devices

All complex PLDs employ Red Square architectures. These are characterised by a small number of large logic blocks surrounding an expanse of fast, direct interconnect. In contrast, all of the FPGAs based on embedded registers employ a Manhattan architecture. Terraced architectures are employed in the one-shot FPGA devices from Actel and Crosspoint. FPGA devices conventionally employ small cells that are connected by sophisticated, and often fragmented, routing resources.

Granularity is an issue that attracts considerable debate. Manufacturers of complex PLDs, having relatively coarse blocks, argue that the simple and direct routing that is possible between a relatively small number of large logic blocks is fast and predictable. It also imposes minimal demands on automatic layout tools and therefore modest tools achieve successful routing in short periods of time. This is in contrast to the situation that is encountered for the complex and fragmented interconnection resources that are necessary for

fine-grained architectures. Also the high functionality of large blocks ensures that most functions can be realised in two levels of logic, and this provides a performance advantage. Alternatively, the vendors of FPGAs, characterised by many fine-grained blocks, argue that they offer higher logic densities and better utilisation of the available resources. In addition, small blocks are preferred for implementing register-intensive designs and therefore offer improved flexibility. Some of the most recent FPGA architectures address these conflicting requirements by providing hybrid logic blocks having a mixed medium/coarse-grained nature. Clearly there is no consensus of opinion and one technology will not be the most appropriate for all applications.

Generally the style of reconfiguration falls into two groups: Static reconfiguration and dynamic reconfiguration.

Static reconfiguration means that a system loads the programming data to the FPGA while the system is not actually operating. The static reconfiguration is referred as a compile-time reconfiguration. In this style, the programming data are loaded into the FPGA only once, in a phase previous to normal operation, this phase is called the configuration phase. Because configuration time can normally be ignored, the configuration circuit is simple; i.e. simplicity is preferred over speed.

On the contrary, **dynamic reconfiguration** can swap the logic contents of an FPGA during system operation without disturbing the operation of the overall system. There are two types of dynamic reconfiguration depending on the FPGA configuration circuitry. One is **Full Chip Reconfiguration**, in which the system reprograms the entire FPGA even if the new configuration data differs only slightly from the original one. The other is **Partial Reconfiguration**, in which the system reprograms only the section of the FPGA that requires changes while the rest of the circuit remains the same. Table 3.2 summarises the reconfiguration styles.

	Static Reconfiguration (Full Chip)	**Dynamic Reconfiguration**	
		Full Chip	Partial
When	Compile time	During system operation	
Time to Reprogram	Don't care	Millisecconds	Nano to microseconds
Reprogram Area	Entire FPGA	Entire FPGA	Portion of FPGA
Purpose	• Definitive system alteration • Hardware upgrade	• Logic swap • Performance enhancement	• Time-multiplexing of hardware • Evolvable HW

Table 3.2 Reconfiguration styles for FPGAs

3.5.2 FPGAs Applications

With the development of FPGAs there are now opportunities to implement systems quite different from those designed with other technologies. Some of these new opportunities, especially those of multi-FPGA systems, will be discussed next.

When FPGAs were first introduced they were primarily considered to be just another form of gate array. While they had lower speed and capacity, and had a higher unit cost, they did not have the large start-up costs and lead times necessary for ASICs. Thus, they could be used for implementing random logic and glue logic in small volume systems with non-aggressive speed and capacity demands. If the capacity of a single FPGA was not enough to handle the desired functionality, multiple FPGAs could be included on the board, distributing the functionality between these chips.

With the advances in technology, FPGAs are nowadays more than just slow, small gate arrays. The critical feature of SRAM-based FPGAs is their in-circuit reprogrammability. Since their programming can be changed quickly, without any rewiring or refabrication, they can be used in a much more flexible manner than standard gate arrays. One example of this is multi-mode hardware. For example, when designing a digital tape recorder with error-correcting codes, one way to implement such a system is to have separate code generation and code-checking hardware built into the tape machine. However, there is no reason to have both of these functions available simultaneously, since when reading from the tape there is no need to generate new codes, and when writing to the tape the code checking hardware will be idle. Thus, it is possible to have an FPGA in the system, and have two different configurations stored in ROM, one for reading and one for writing. In this way, a single piece of hardware handles different functions. There have been several multi-configuration systems based on FPGAs, including the tape machine, generic printer and CCD camera interfaces, and pivoting monitors with landscape and portrait configurations [Hau97, Xil94, Cas97, Tem94].

While the previous uses of FPGAs still treat these chips purely as methods for implementing digital logic, there are other applications where this is not the case. A system of FPGAs can be seen as a computing substrate with somewhat different properties than standard microprocessors. The reprogrammability of the FPGAs allows downloading algorithms onto the FPGAs, and changing these algorithms just as general-purpose computers can change programs. This computing substrate is different from standard processors in that it provides a huge amount of fine-grain parallelism. The instructions are quite simple, in the order of a single five bit input, one bit output function. Moreover, while the instruction stream of a

microprocessor can be arbitrarily complex, with the function computed by the logic changing on a cycle by cycle basis, the programming of an FPGA is in general held constant throughout the execution of the mapping (exceptions to this include techniques of run-time reconfigurability described below). Thus, if a variety of different functions in a mapping are needed, a microprocessor executes them temporally, i.e. with different functions executed during different cycles; whereas an FPGA-based computing machine achieves variety spatially, i.e. letting different logic elements compute different functions. This means that microprocessors are superior for complex control flow and irregular computations, while an FPGA-based computing machine can be superior for data-parallel applications, where a huge amount of data must be acted on in a very similar manner.

There have been several computing applications where a multi-FPGA system has delivered the highest performance implementation. An early example is generic string matching on the Splash machine [Gok90]. Here, a linear array of Xilinx 3000 series FPGAs was used to implement a systolic algorithm to determine the "edit distance" between two strings. The edit distance is the minimum number of insertions and deletions necessary to transform one string into another, so the strings "flea" and "fleet" would have an edit distance of 3 (delete "a" and insert "et"). A dynamic-programming solution to this problem can be implemented in the Splash system as a linear systolic circuit, with the strings to be compared flowing in opposite directions through the linear array. Processing can occur throughout the linear array simultaneously, with only local communication necessary, producing a huge amount of fine-grain parallelism. This is exactly the type of computation that maps well onto a multi-FPGA system. The Splash implementation was able to offer an extremely high performance solution for this application, achieving performance approximately 200 times faster than supercomputer implementations.

Table 3.3 presents a list of applications where a multi-FPGA system has offered the highest performance solution:

• Long multiplication	• Travelling salesman problem	• Speech recognition
• Real-time pattern recognition	• Genetic optimisation	• Genetic database searches
• Stereo matching in stereo vision	• Region detection and labelling	• Differential equations solvers
• Monte Carlo algorithms	• Cryptography	• Modular multiplication

Table 3.3 High-performance applications of FPGAs

One of the most successful uses for FPGA-based computation is in ASIC logic emulation. The idea is that the designers of a custom ASIC need to make sure that the circuit they designed correctly implements the desired functionality. Software simulation can perform these checks, but does so slowly. In logic emulation the circuit to be tested is instead mapped onto a multi-FPGA system, yielding a solution several orders of magnitude faster than software simulation.

An emerging application of FPGA-based computing is the training and execution of neural networks [Dye95]. A neural network is a powerful computational model based on the structure of neurons in the brain. These systems have proven effective for tasks such as pattern recognition and classification. Neural network implementations take advantage of FPGA's reprogrammability by changing the chip's programming over time, much as a standard processor context-switches to a new program. However, it is possible to make more aggressive use of this ability to develop new types of applications.

The FPGA can be viewed as a demand-paged hardware resource, yielding "**virtual hardware**" similar to virtual memory in today's computers. In such systems (usually grouped under the term "**dynamically reconfigurable**" or "**run-time configurable**"), an application will require many different types of computations, and each of these computations has a separate mapping to the programmable logic. For example, an image processing application for object thinning may require separate pre-filtering and thresholding steps before running the thinning operation, each of which could be implemented in a separate FPGA mapping. Although these mappings could be spread across multiple FPGAs, these steps must take place sequentially, and in a multi-FPGA system only one mapping would be actively computing at a time. Run-time configuration saves hardware by reusing the same resource. Because of these advantages, there has been a significant amount of work on run-time reconfigurable systems, applications, and support tools, e.g. [Bli91, Cas97, Gok90, Jay93].

Researchers at the MIT have proposed an FPGA that stores multiple configurations in a series of memory banks. In a single clock cycle, which is of the order of tens or hundreds of nanoseconds, the chip could swap one configuration for another configuration without erasing partially processed data [Deh95].

At Brigham Young University configurable computing has been used to build a dynamic instruction set computer (DISC), which effectively marries a microprocessor to an FPGA and demonstrated the potential of automatic reconfiguration using stored configurations. As a program runs, the FPGA requests reconfiguration if the designated circuit is not resident.

DISC allows a designer to create and store a large number of circuit configurations and activate them much as a programmer would initiate a call to a software subroutine in a microprocessor [Hut95].

The Colt Group of Virginia Polytechnic Institute and State University, is investigating a run-time reconfiguration technique called Wormhole that lends itself to distributed computing. The unit of computing is a stream of data that creates custom logic as it moves through the reconfigurable hardware [Ath97].

Researchers at the University of California at Berkeley are developing systems that combine a microprocessor and an FPGA. Special compiler software would take ordinary program code and automatically generate a combination of machine instructions and FPGA configurations for the fastest overall performance. This approach takes advantage of opportunities to integrate a processor and an FPGA on a single chip.

In recent years FPGAs have become the cornerstone of Evolvable Hardware. Adrian Thompson and his team at Sussex University have used Xilinx FPGAs to evolve digital circuits [Tho96b]. By using the configuration bits of an FPGA as the population of a genetic algorithm, Thompson was able to "evolve" a frequency discriminator [Tho97]. This work is of particular importance because, for the first time, the parasitic characteristics of electronic circuits were used to provide useful work. However, the behaviour of the evolved circuit remained beyond explanation until recently [Tho99].

Another remarkable use of FPGAs in the field of evolvable hardware can be found in the work done by Moshe Sipper from the EPFL, Switzerland. Sipper succeeded in intrinsically evolving the behaviour of a linear cellular automaton [Sip97a]. Intrinsically means that both, the aiming task and the genetic algorithm were running concurrently in the FPGA.

The Logic Systems Laboratory in Lausanne, Switzerland originally proposed Embryonics as a new family of fault-tolerant FPGAs inspired by nature. The main idea is to have reconfigurable systems with the ability to perform self-diagnosis and self-reconfiguration with no assistance from an external agent. This architecture will be extensively revised in the next chapter.

3.5.3 The future of configurable computing

Configurable computing is still an extremely young field. Although this approach was proposed since the late 1960s, the first demonstrations did not occur until a few years ago and current FPGAs with up to 100,000 logic elements still do not come close to exploiting

the full possibilities of the technique [Vil97]. Future FPGAs will be much larger. As with many other integrated circuits, the number of elements on a single FPGA has doubled roughly every 18 months. Before the decade of the 1990's is out, it is expected to see FPGAs that have a million logic elements. Such chips will have much broader application, including highly complex communications and signal-processing algorithms.

Academic researchers and manufacturers are overcoming numerous other design limitations that have hindered the adoption of configurable computing. Not all computations can be implemented efficiently with today's FPGAs: they are well suited to algorithms composed of bit-level operations, such as pattern matching and integer arithmetic, but they are ill suited to numeric operations, such as high-precision multiplication of floating-point numbers. Dedicated multiplier circuits such as those used in microprocessor and digital signal chips can be optimised to perform more efficiently than multiplier circuits constructed from configurable logic blocks in an FPGA. Furthermore, FPGAs currently provide very little on-chip memory for storage of intermediate results in computations; thus, many configurable computing applications require large external memories. The transfer of data to and from the FPGA increases power consumption and may slow down the computations.

FPGAs will never replace microprocessors for general-purpose computing tasks, but the concept of configurable computing is likely to play a growing role in the development of high-performance computing systems. The computing power that FPGAs offer will make them the devices of choice for applications involving algorithms in which rapid adaptation to the input is required.

In addition, the line between programmable processors and FPGAs will become less distinct: future generations of FPGAs will include functions such as increased local memory and dedicated multipliers that are standard features of today's microprocessors; there are also next-generation microprocessors under development whose hardware supports limited amounts of FPGA-like reconfiguration. Indeed, just as computers connected to the Internet can now automatically download special-purpose software components to perform particular tasks, future machines might download new hardware configurations, as they are needed. Computing devices 10 years from now may include a strong mix of software-programmable hardware and hardware-reconfigurable logic [Vil97].

Finally, while existent FPGA cells execute basic logic operations, research is being done to design FPGAs whose cells can execute mathematical operations; e.g. adders or multipliers [Tis99]. These circuits require fewer resources, while facilitating the hardware implementation of complex operations [Mar99].

3.6 Binary Decision Diagrams

Boolean algebra forms a cornerstone of computer science and digital system design. Many applications in digital logic design and testing, artificial intelligence, and graph analysis can be expressed as a sequence of operations on Boolean functions. Such applications would benefit from efficient algorithms for representing and manipulating Boolean functions symbolically. Unfortunately, many of the tasks one would like to perform with Boolean functions require solutions to NP-complete or co-NP-complete problems. For example, testing whether there exists any assignment of input variables such that a given Boolean expression evaluates to 1 (satisfiability), or two Boolean expressions denote the same function (equivalence) [Bry86]. Consequently, all known approaches to perform these operations require, in the worst case, an amount of computer time that grows exponentially with the size of the problem. This makes it difficult to compare the relative efficiencies of different approaches to represent and manipulate Boolean functions. In the worst case, all known methodologies perform as poorly as the naïve approach of representing functions by their truth tables and defining all of the desired operations in terms of their effect on truth table entries. In practice, by utilising alternative representations one can often avoid these exponential computations.

A variety of methods have been developed for representing and manipulating Boolean functions. Those based on classical representations such as truth tables, Karnaugh maps, or canonical sum-of-products form are quite impractical; every function of n arguments has a representation of size 2^n.

More practical approaches use representations that, at least for many functions, are not of exponential size, e.g. reduced sum of products. These representations suffer from several drawbacks. First, certain common functions still require representations of exponential size, e.g. the even and odd parity functions serve as worst case examples in all these representations. Second, while a certain function may have a reasonable representation, performing a simple operation, such as taking the complement, could yield a function with exponential representation. Finally, none of these representations are **canonical forms**, i.e. a given function may have many different representations. Consequently, testing for equivalence or satisfiability can be quite difficult.

Due to these characteristics, most programs that process a sequence of operations on Boolean functions have rather erratic behaviour. They proceed at a reasonable pace, but then suddenly "blow up", either running out of storage or failing to complete an operation in a reasonable amount of time [Bry86].

A **binary decision tree or diagram** (BDD) is a model of the evaluation of a Boolean function, wherein the value of a variable is determined and the next action (to choose another variable to evaluate or to output the value of the function) is chosen accordingly. BDDs find many applications in fields such as: decision table programming [Met77], databases [Han77], pattern recognition [Bel78], taxonomy and identification [Pay80], machine diagnosis [Cha70], switching theory [Lee59], and analysis of algorithms [Wei77]. Due to this broad applicability, results about BDDs are dispersed throughout the literature in fields such as biology, computer science, information theory, and switching theory.

BDD notation was introduced by Lee [Lee59] and further popularised by Akers [Ake78]. Bryant placed restrictions on the ordering of decision variables, which enabled the development of algorithms for manipulating the representations in a more efficient manner. Bryant representation was called the **Ordered BDD** or **OBDD**, and offered the advantage of being canonical. This property has several important consequences [Bry92]: Functional equivalence can be easily tested. A function is satisfiable if its OBDD representation does not correspond to the single terminal vertex labelled 0. Any tautological function must have the terminal vertex labelled 1 as its OBDD representation. If a function is independent of variable x, then its OBDD representation cannot contain any vertices labelled by x. Thus, once OBDD representations of functions have been generated, many functional properties become easily testable.

3.6.1 Construction of a Binary Decision Diagram

A BDD represents a Boolean function as a rooted, directed acyclic graph. As an example, figure 3.10 illustrates a representation of the function $f(x_1,x_2,x_3)$ defined by the truth table given on the left, for the special case where the graph is actually a tree.

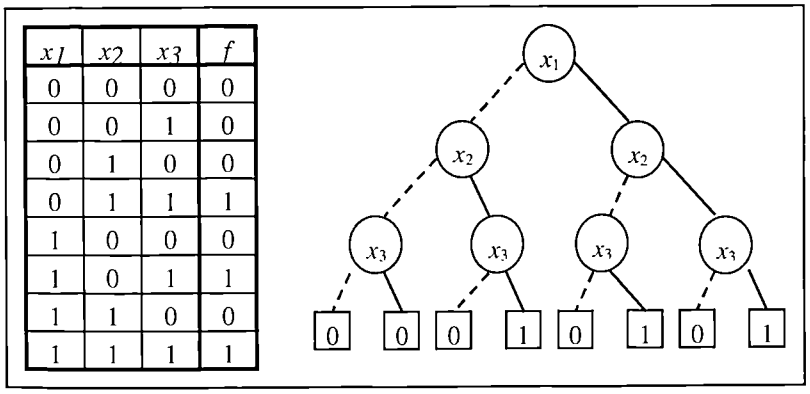


Figure 3.10 Truth table and decision tree of a Boolean function

Each non-terminal vertex v is labelled by a variable $var(v)$ and has arcs directed toward two children: $lo(v)$ (shown as a dashed line) corresponding to the case where the variable is assigned 0, and $hi(v)$ (shown as a solid line) corresponding to the case where the variable is assigned 1. Each terminal vertex is labelled 0 or 1. For a given assignment to the variables the value yielded by the function is determined by tracing a path from the root to a terminal vertex, following the branches indicated by the values assigned to the variables. The function value is then given by the terminal vertex label. Due to the way the branches are ordered in figure 3.10, the values of the terminal vertices, read from left to right, match those in the truth table, read from top to bottom.

In order to construct an Ordered Binary Decision Diagram (OBDD) a total ordering over the set of variables is imposed, and require that for any vertex u, and non-terminal child v, their respective variables must be ordered $var(u) < var(v)$. In the decision tree of figure 3.10, for example, the variables are ordered $x_1 < x_2 < x_3$. In principle, the variable ordering can be selected arbitrarily because the algorithm will operate correctly for any ordering. In practice, selecting a satisfactory ordering is critical for an efficient symbolic manipulation.

Three transformation rules are defined such that the function represented by these graphs is not altered [Bry92]:

1. **Remove Duplicate Terminals**. Eliminate all but one terminal vertex with a given label and redirect all arcs into the eliminated vertices to the remaining one.
2. **Remove Duplicate Non-terminals**. If non-terminal vertices u and v have $var(u)= var(v)$, $lo(u)= lo(v)$, and $hi(u)= hi(v)$, then eliminate one of the two vertices and redirect all incoming arcs to the other vertex.
3. **Remove Redundant Tests**. If non-terminal vertex v has $lo(v)= hi(v)$, then eliminate v and redirect all incoming arcs to $lo(v)$.

Starting with any BDD satisfying the ordering property, we can reduce its size by repeatedly applying the transformation rules. The term **OBDD** refers to a maximally reduced graph that obeys some ordering. For example, figure 3.11 illustrate the reduction of the decision tree shown in figure 3.10 into an OBDD.

Applying the first transformation rule to figure 13.10 reduces the eight terminal vertices to two (Fig.3.11a). Applying the second transformation rule eliminates two of the vertices having variable x_3, and the arcs to terminal vertices with labels 0 (lo) and 1 (hi) (Fig.3.11b). Applying the third transformation rule eliminates two vertices: one with variable x_3 and one

with variable x_2 (Fig.3.11c). In general, the transformation rules must be applied repeatedly, since each transformation can expose new possibilities for further ones.

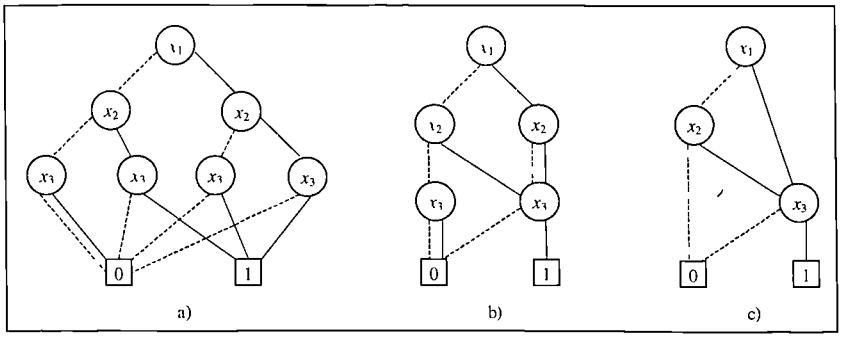


Figure 3.11 Reduction of decision tree into OBDD

As figures 3.10 and 3.11 illustrate, it is possible to construct the OBDD representation of a function given its truth table by constructing and reducing a decision tree. However, this approach is practical only for functions of a small number of variables, since both the truth table and the decision tree have size exponential in the number of variables. The form and size of the OBDD representing a function depends on the variable ordering. Most applications using OBDDs choose some ordering of the variables at the outset and construct all graphs according to this ordering. This ordering is chosen manually or by heuristic analysis of the particular system to be represented. For example, several heuristic methods have been devised that generally derive a good ordering for variables representing the primary inputs [Mor82]. Others have been developed for sequential-system analysis [Jeo91]. State of the art techniques propose the use of evolutionary methods to find an optimal variable ordering for a given function [Sak97]. Note that these heuristics do not need to find the best possible ordering since the ordering chosen has no effect on the correctness of the results. As long as an ordering can be found that avoids exponential growth, operations on OBDDs remain reasonably efficient.

3.6.2 Implementation of Binary Decision Diagrams

BDDs and OBDDs can easily be programmed. Lee called the result *decision programs* [Lee59], and has suggested a universal instruction type which implements the evaluation process taking place at an internal node:

$$L: i, g0, g_1, \ldots, g_{m-1}$$

Where L is a label, i identifies variable x_i, and g_k (used only when $x_i=k$) is either a value (if the restriction for $x_i=k$ is a constant) or a label. Such an instruction is executed by testing variable x_i and upon finding its value, say $x_i=k$, taking the corresponding action g_k; that is,

either transferring control to the instruction labelled g_k, or assigning to the function the value g_k. Thus, to each node of the diagram there corresponds one instruction in the program. Cerny has investigated a special architecture for the execution of such programs [Cer79].

OBDDs can also be implemented in hardware as multiplexer trees and networks [Cer79]. In a multiplexer tree, each internal tree node is represented as a 2-1 multiplexer controlled by the node variable, and each leaf is implemented as a constant logical value (wired at 0 or wired at 1); the interconnection scheme is that of the OBDD. The evaluation of a function then proceeds from the "leaves" (the constant values) to the root multiplexer. The function variables, used as control variables, select a unique path from the root to one leaf, and the value assigned to that leaf propagates along the path to the output of the root multiplexer.

Consider the OBDD of figure 3.11c. Its diagram can be implemented either by a decision program (figure 3.12a) (where letters are used for labels to distinguish them from values), or by a multiplexer network (figure 3.12b).

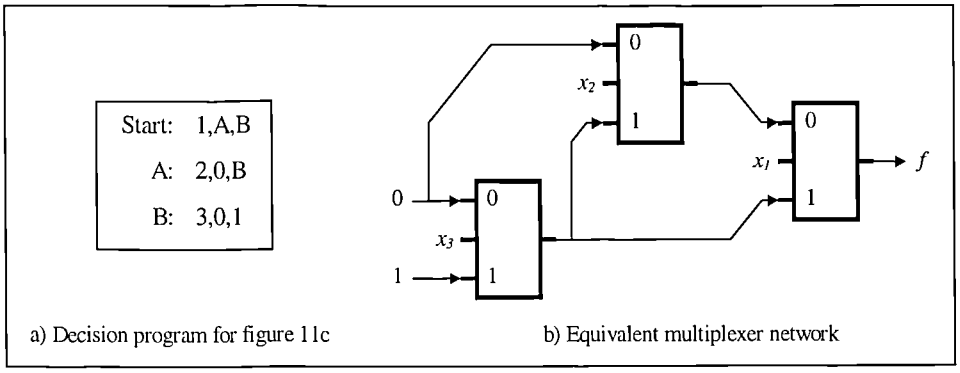


Figure 3.12 Decision program and multiplexer network for figure 3.11c

Note that the number of multiplexers used in a network is precisely the number of instructions of an equivalent decision program. Similarly, the maximum delay though a network is proportional to the maximum execution time of an equivalent program, both being dependent upon the length of the longest path through the diagram.

3.6.3 Application of OBDDs

The use of OBDDs in digital-system design, verification and testing has gained widespread acceptance. In this section a few areas of application are described.

Verification

OBDDs can be applied directly to the task of testing the equivalence of two combinational logic circuits. This problem arises when comparing a circuit to a network derived from the

system specification, or when verifying that a logic optimiser has not altered the circuit functionality [Bry86].

Design error correction

Going further to simply detect the existence of errors in a logic design, researchers have developed techniques to automatically correct a defective design. This involves considering some relatively small class of potential design errors, such as a single incorrect logic gate, and determining if any variant of the given network could meet the required functionality [Mad89]. This analysis demonstrates the power of the quantification operations for computing projections, in this case projecting out the primary input values by universal quantification.

Sensitivity analysis

A second class of applications involves characterising the effects of altering the signal values on different lines within a combinational circuit. That is, for each signal value s, the Boolean difference for every primary output with respect to s is computed. This analysis can be performed symbolically by introducing "signal line modifiers" into the network, i.e. for each line that would normally carry a signal value s, this value is selectively altered to be the complement of s (s') under the control of a Boolean value P by computing s'= s XOR P. The conditions under which a particular output of the circuit is sensitive to the value on a signal line can be determined by comparing the outputs of the original and altered circuits. One application of this sensitivity analysis is automatic test generation. A second application is in the area of combinational logic optimisation [Cho89].

Probabilistic analysis

Recently, researchers have devised a method for statistically analysing the effects of varying circuit delays in a digital circuit. This application of OBDDs is particularly intriguing, since conventional wisdom would hold that such an analysis requires evaluation of real-valued parametric variations and hence could not be encoded with Boolean variables. Consider a logic gate network in which each gate has a delay given by some probability distribution. This circuit may exhibit a range of behaviours, some of which are classified as undesirable. The yield is then defined as the probability that these behaviours do not occur. One simple analysis would be to treat the waveform probabilities for all signals as if they were independently distributed. The behaviour of each gate output can be computed according to the gate function and input waveforms. To solve this problem through symbolic Boolean analysis two restrictions must be made. First, all circuit delays must be integer valued, and

hence transitions occur only at discrete time points. Second, the delay probabilities for a gate must be multiples of a value $1/k$, where k is a power of 2. Given a Boolean function representing the conditions under which some event occurs, we can compute the event probability by computing the density of the function, i.e. the fraction of variable assignments for which the function yields 1.

Other applications

Historically, OBDDs have been applied mostly to tasks in digital-system design, verification and testing. However, their use has recently spread into other application domains. For example, the fixed-point techniques used in symbolic-state machine analysis can be used to solve a number of problems in mathematical logic and formal languages, as long as the domains are finite. Researchers have also shown that problems from many application areas can be formulated as a set of equations over Boolean algebras that are solved by a form of unification.

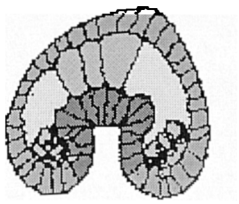
In the area of artificial intelligence, researchers have developed a truth maintenance system based on OBDDs [Mad91]. They use an OBDD to represent the database, i.e. the known relations among the elements. They have found that by encoding the database in this form, the system can make inferences more readily than with the traditional approach of simply maintaining an unorganised list of known facts. For example, determining whether a new fact is consistent with or follows from the set of existing facts involves a simple test for implication.

3.7 Summary

The ideas and technologies that give shape to the embryonics project have been presented in this chapter. Like any other bio-inspired system, embryonics has both biological and technological foundations and, although many of these ideas are quite dissimilar to each other, every effort has been made to present them in a clear and ordered manner.

Embryonics applies mechanisms that take place during the development of embryos to the design of field-programmable processor-arrays. The result is a fault-tolerant cellular architecture capable of implementing, in its present stage, any logic function represented as an ordered binary decision diagram.

The concepts and technologies presented in this chapter constitute the background information that supports the embryonics project. The following chapter proposes a detailed implementation of an embryonic architecture.



CHAPTER 4

ARCHITECTURE OF AN EMBRYONIC SYSTEM

This chapter presents a detailed description of the MUXTREE embryonic architecture. Section 4.2 presents the block diagram of a generic embryonic cell as well as a description of each one of its constituent blocks. Section 4.3 discuses the built-in self-test techniques employed to endow the cell with fault tolerance, and the cost associated with it. Section 4.4 presents three examples of the use of embryonic arrays. Resilience to faults is verified by means of simulation.

4.1 Introduction

Previous chapters presented the biological and technological background that sustains the embryonics project. During the development of the project several implementations of the embryonics architecture have been proposed [Man98a, Ort99a]. In this chapter **MUXTREE,** a particular implementation of the embryonics architecture, is presented.

MUXTREE designates an embryonic cell whose processing element is a selector or multiplexer [Ort98a, Tem97]. Multiplexers have the characteristic of being able to implement any node from an ordered binary decision diagram (OBDD), which in turn can represent any combinational or sequential logic function [Ake78, Cer79, Lia92].

A top-down methodological approach was followed during the design of the cell, while a bottom-up approach was followed for implementing the functional blocks. The cell was implemented and simulated in the Viewlogic's Workview® suite. The resulting architecture resembles that of Actel's commercial FPGAs [Act95].

In section 4.2 the general block diagram of MUXTREE's fundamental cell is presented. Every constituent module of the block diagram will be described in the subsections that follow. The schematic diagrams of the MUXTREE cell can be found in appendix A.

Section 4.3 presents the built-in self-test techniques employed to provide the MUXTREE with self-diagnosis capabilities. The cost associated with these improvements is discussed in subsection 4.3.4. The reconfiguration mechanisms that are activated when a fault is detected in any of the cells are explained in section 4.3.5.

Section 4.4 introduces a methodology to implement logic applications using embryonic arrays. This methodology is used in three simple applications: a voter circuit, a 3-bit up-down counter, and a programmable frequency divider. Simulations showing the resilience to injected faults are presented for each example.

4.2 The Embryonics Architecture

An embryonic array is a regular array of interconnected embryonic cells. In resemblance to natural cellular systems, every cell in an embryonic array performs the same basic operation regardless of the particular logic function it is involved with. Each cell interprets one of the configuration registers allocated in its memory (genome) to perform the logic operations needed for the correct implementation of the system's specification. The configuration register that is selected depends on the position of the cell within the array, distinguished by a set of co-ordinates. The co-ordinates are calculated locally from those of the nearest neighbouring cells.

Figure 4.1 shows the basic architecture of a generic embryonic system.

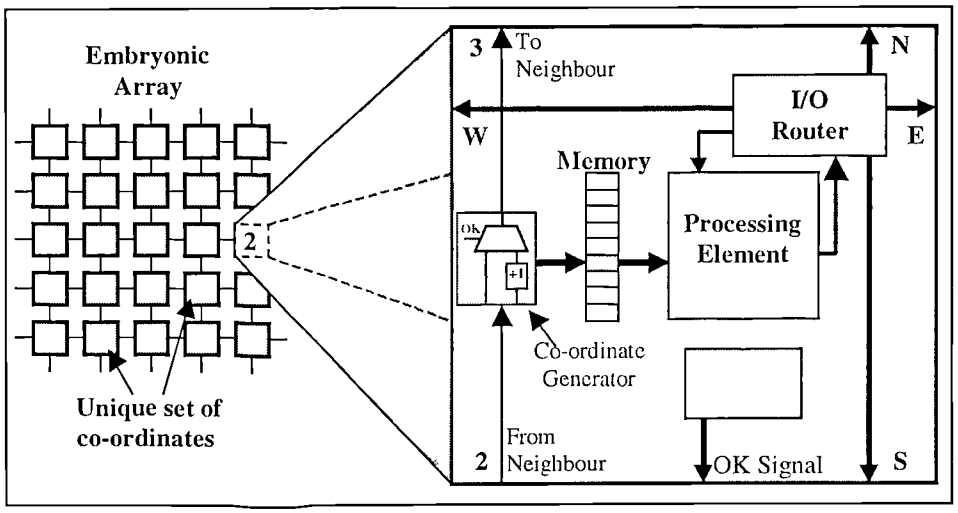


Figure 4.1 Basic Components of an Embryonic System

In the MUXTREE architecture, each cell is distinguished by one co-ordinate. The processing element is a multiplexer whose functionality and input/output routing are controlled by a configuration register that is selected by the cell's co-ordinate.

Digital data are transmitted from one cell to its neighbours through a direct North-East-West-South (**NEWS**) connection. The I/O router also propagates information over the entire array by means of a routing network that acts as a "virtual bus". The I/O router is controlled by one section of the corresponding configuration register.

Embryonic cellular arrays share the following properties with their biological counterparts [Mar96]: Multicellular organisation (every cell has a unique set of co-ordinates), cellular differentiation (each cell performs a unique function), and cellular division (the genome is copied from mother to daughter cells).

The architecture shown in figure 4.1 presents the following advantages:

- It is highly regular, which simplifies its implementation on silicon.
- The actual function of the processing element is independent from the function of the remaining blocks. This modularity has been exploited to produce families of embryonic field programmable arrays, each one offering different functionalities and complexity. For example, the MUXTREE architecture implements a binary selection function [Tem97], whereas the MICTREE family follows a microprogrammed approach [Man98a].
- Provided the architecture of the processing element is kept simple, it would be possible to include built-in self test (BIST) logic to provide self-diagnosis without excessively incrementing the silicon area [Lal85, Tur90].

The following subsections describe the blocks that make up the MUXTREE cell.

4.2.1 Memory Architecture

Each cell must have enough memory to maintain a copy of the configuration registers of all the cells in the corresponding column for the row-elimination strategy to be achieved. One extra register must be provided to allocate the configuration that is selected when faults are detected (transparent configuration).

MUXTREE cells require 17 bits to be fully configured. The definition of these bits is given in section 4.2.5. In order to simplify the genome's downloading process, memory is designed using a serial-input parallel-output shift register. During downloading external logic enables one column in the array at one time so that information flows into all the cells in that column

simultaneously. This approach offers the possibility during normal operation of re-loading the configuration registers of any column without interfering with the configuration stored in other columns. Following the DNA resemblance, columns in the array can be considered as chromosomes. This chromosomic approach is particularly attractive if the architecture is to be evolved using genetic algorithms.

For an array where columns have *n* cells each, the total amount of configuration bits required on each embryonic cell is:

(*n*+1 registers per cell) * (17 bits per register) + (2 bits for diagnosis) = (17n + 19) bits

Figure 4.2 shows the memory structure for an array containing 16 cells per column. Note that the size of the memory is independent of the number of columns utilised for a particular application.

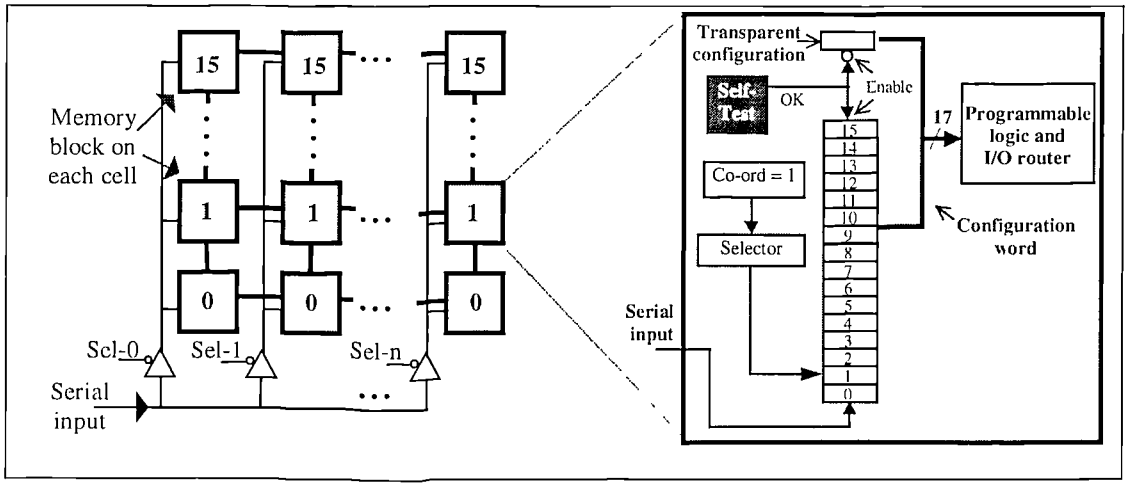


Figure 4.2 Embryonic cell's memory architecture

4.2.2 Co-ordinate Generator

Each cell in an embryonic array has a co-ordinate that is unique in the corresponding column. Co-ordinates are received from the south neighbours. Cells in the south edge of the array are hard-wired to co-ordinate 0. If one of the cells self-diagnoses faulty through the mechanisms exposed in section 4.3, then it becomes transparent and propagates its co-ordinate without increment. Therefore, the north neighbour takes the faulty cell's co-ordinate and consequently its function.

Each cell must generate a co-ordinate for its north neighbour. The value of the co-ordinate issued depends on the status of the cell that generates it. If the cell is non-faulty, then its co-ordinate is incremented by 1 and the resulting value propagated north. If the cell self-

diagnoses faulty, then it becomes transparent and propagates the input co-ordinate to the output. Figure 4.3 illustrates the process of co-ordinates generation.

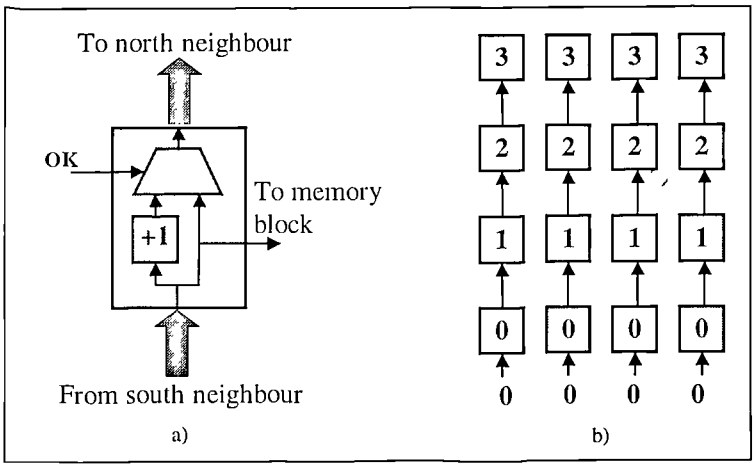


Figure 4.3 a) Co-ordinates generator and b) Co-ordinates generation

No "code" is being communicated around the array when reconfiguration takes place, only Boolean signals are passed between cells. These actions are carried out by combinational logic, therefore, the time required to reconfigure a complete array is very short, depending only on the number of cells per column and the propagation delays of the gates involved.

Setting-up an embryonic array implies two phases: the configuration phase, in which the co-ordinates are calculated and the genome is downloaded; and the operational phase, in which the array performs the desired function. During the configuration phase, co-ordinates are ignored and no output in the whole array is valid. When the last configuration bit has been shifted into the array the operational phase begins, each cell in the array selects a configuration register and outputs become valid.

4.2.3 Processing Element

The processing element in the MUXTREE performs a 2-1 multiplexer function. This is basically a 2-input selector, where each input can be selected from one of 8 possible sources. The output can be registered and fedback so that the implementation of sequential logic is possible. Figure 4.4 shows the architecture of this block.

The selection element shown in figure 4.4 allows many input-output combinations by programming the configuration bits (labels in bold). This selection capability, in conjunction with the I/O router, allows the implementation of Ordered Binary Decision Diagrams (OBDDs) of any size, provided the number of cells in the array is sufficient [Lia92].

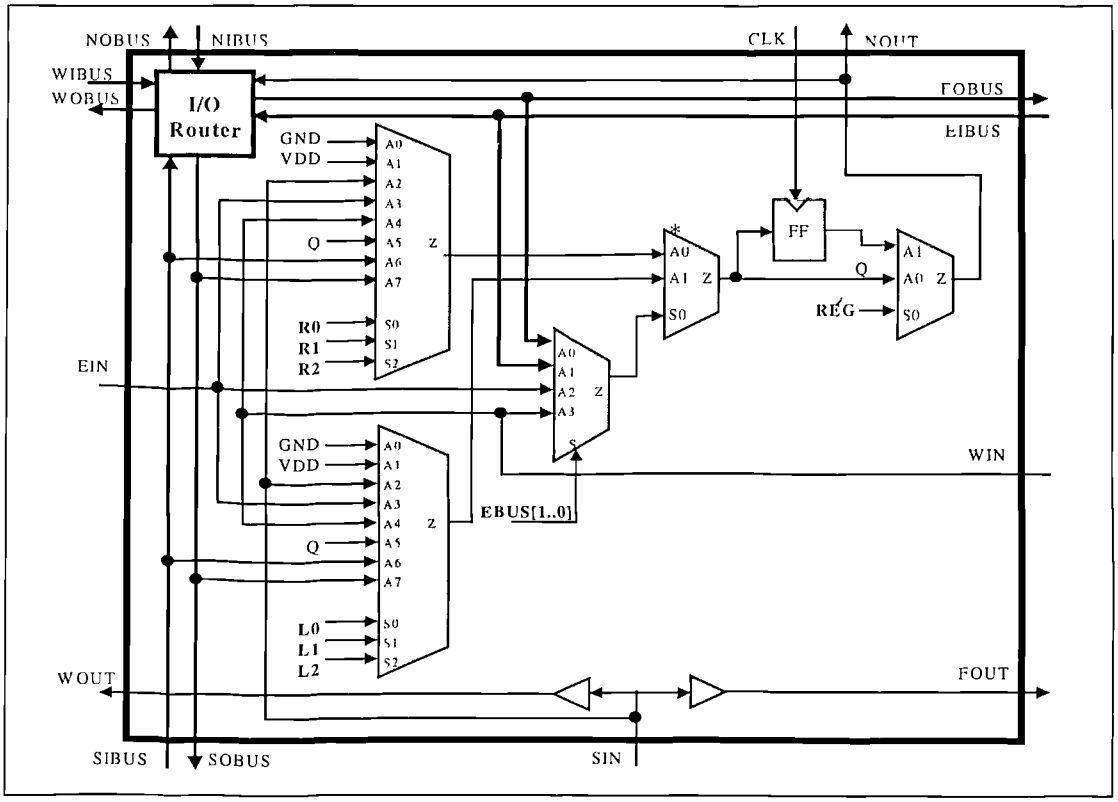


Figure 4.4 Architecture of processing element inside MUXTREE cell

There are two types of data input for this module: those that are propagated naturally, i.e. without any kind of routing configuration; and those whose source and destination are selected via the configuration register and the I/O router module. EIN, WIN, SIN, NOUT, WOUT and EOUT belong to the first group, while EOBUS, EIBUS, SIBUS and SOBUS belong to the second group. Controlled signals allow the interchange of information between non-neighbouring cells. Prefixes N, E, W and S indicate which neighbour is either receiving or transmitting the corresponding signal, for example NOUT is the non-controlled output going to the north neighbour, while EIBUS means the controlled input coming from the neighbour on the east.

L2:0 and **R2:0** select one input out of eight on their respective multiplexers. It is possible to select 0 or 1 as the signal to be propagated in order to facilitate the implementation of the terminal nodes of the OBDDs. Notice that the registered output Q is fedback on input A5 of each multiplexer, this allows the implementation of sequential circuits. The REG bit in the configuration register will determine if the output is combinational or sequential. The selection input for the main multiplexer element (marked with a star in figure 4.4) can also be selected from four of the signals controlled by the I/O router. The value of bits **EBUS1:0** on the configuration register determine whether EOBUS, EIBUS, EIN or WIN will select the block's output. The south input SIN is propagated through both EOUT and WOUT outputs.

It is necessary to bear in mind that during the operational phase each cell is part of a cellular array, therefore complexity is achieved not by the function performed by a single cell but by the simultaneous interaction of all cells in the array [Lan84].

4.2.4 Input/Output Router

In a conventional cellular array communications among cells are restricted by the NEWS interconnection, i.e. the output generated by a particular cell can only be propagated to the nearest neighbours. To overcome this limitation the I/O router provides additional paths so that information can be propagated not only to the nearest neighbours, but also to more distant ones. Figure 4.5 shows the way information is routed in this block. Labels in bold represent the selection bits stored in the configuration register.

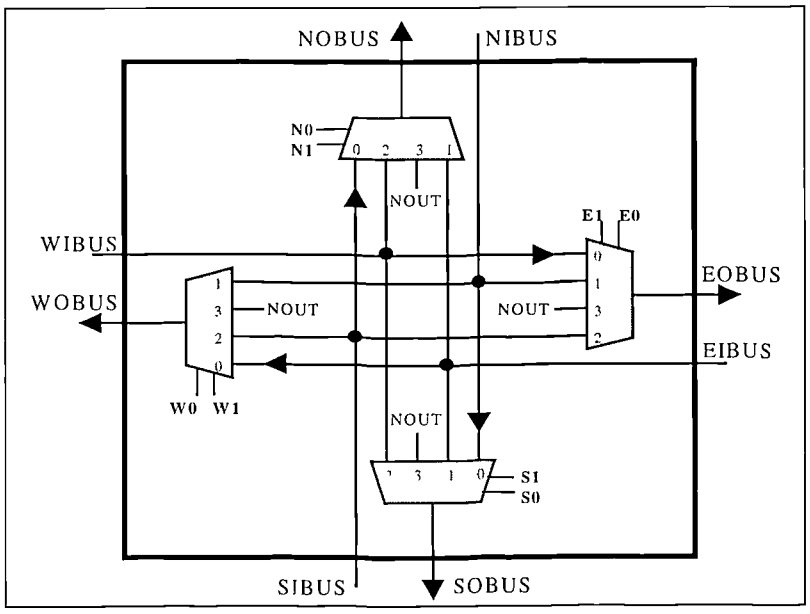


Figure 4.5 I/O Router

Figure 4.5 shows the various paths any input can follow. Configuration bits **N1:0**, **S1:0**, **E1:0** and **W1:0** select one of 4 possible outputs on the corresponding selector.

Inside every router in the array, NOUT is the output coming from the corresponding selection element. If necessary, this signal can be propagated simultaneously through all output virtual buses. I/O router inputs can be sent to any direction except the one they came from, e.g. NIBUS cannot be returned through NOBUS. The I/O router allows fast long-distance interconnections between cells.

When a fault is detected and reconfiguration takes place, the I/O router assumes a default configuration in which data propagate following a straight line; e.g. SIN is routed to NOUT.

4.2.5 Configuration register

Figure 4.6 shows the contents of the configuration register.

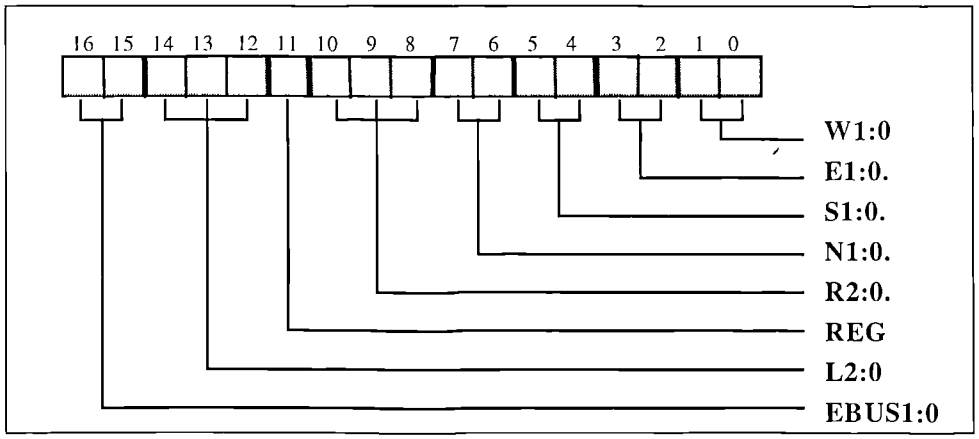


Figure 4.6 Configuration register

EBUS1:0- Determines whether the selection input for the main multiplexer will be taken from EOBUS (**EBUS**= 0), EIBUS (**EBUS**= 1), EIN (**EBUS**= 2) or WIN (**EBUS**= 3).

L2:0, R2:0- The left (L) and right (R) inputs for the main multiplexer are selected according to the value of these bits. There are eight possible input signals and one is selected by the combination of these bits.

REG- If this bit is 1, the output of the logic block becomes the registered output of the main multiplexer element. If it is 0, the direct non-registered output is selected.

N1:0, E1:0, W1:0, S1:0- These bit-pairs select the input that will be propagated on the outputs of the I/O router, they control the output on NOBUS, EOBUS, WOBUS and SOBUS respectively.

Once loaded, the value of the configuration registers remains unchanged. At the end of the configuration process the contents of all the cells in a column are identical. It is the cell's position within the array that determines which configuration register is interpreted. At this level the embryonic array is a static entity. The ability to detect and overcome faults is given by a further mechanism.

4.3 Error Detection and Error Handling

Fault tolerance in processor arrays implies the mapping of a logical array onto a non-faulty physical array; i.e. every logical cell must have a correspondent physical cell. If faults arise, a mechanism must be provided for reconfiguring the physical array so that the remaining

non-faulty cells can still represent the logical array. All reconfiguring mechanisms are based on one of two types of redundancy: time redundancy or hardware redundancy [Che90b].

In time redundancy the tasks performed by faulty cells are distributed among its neighbours. When reconfiguration occurs, processors dedicate some time performing their own tasks and some performing the faulty cells' functions, resulting in some degradation of system's performance. In addition, the algorithm being executed must be flexible enough to allow a simple and flexible division of tasks in real time [For85].

In hardware redundancy physical spare cells and links are used to replace the faulty ones. For this instance, reconfiguring algorithms must optimise the use of spares. In the ideal case an array with N spares must be able to tolerate N faulty cells but, in practice, limitations on the interconnection capabilities of each cell prevent this goal from being achieved.

The majority of hardware redundancy reconfiguration techniques rely on complex algorithms to re-assign physical resources to the elements of the logical array [Neg89]. In most cases, these algorithms are executed by a central processor that performs diagnosis and executes the reconfiguration algorithm [Dut97, For85]. This approach has been demonstrated to be effective, but its centralised nature makes it prone to collapse if the processor in charge of the fault tolerance functions fails. These mechanisms also rely on the designer making a-priori decisions on reconfiguration strategies and data/code movement, which are prone to error and may in practice be less than ideal. Furthermore, the time required by the central controller to perform all these tasks is often prohibitively long and therefore, unsuitable for real-time fault tolerance [All90].

An alternative approach is to distribute the diagnosis and reconfiguration algorithms among all the cells in the array. In this way no central agent is necessary and consequently the reliability and time-response of the system should improve. However, this decentralised approach does tend to increase the complexity of the reconfiguration algorithm and the amount of communication within the network.

The main attractiveness of embryonic arrays resides in their bio-inspired self-reconfiguration abilities, taking away from the designer the need to take complex design decisions. In order to achieve distributed diagnosis and fast reconfiguration in embryonic arrays, every cell must perform both tasks. But cells should also be simple in order to maintain a low failure rate. Therefore a balance between versatility and simplicity must be found. Complex cells would be able to perform more and better diagnosis tasks at the expense of high failure rates. On the other hand, simple cells would have long mean time between failures (MTBF), but their diagnosis and processing capabilities had to be necessarily restricted.

The following sections describe the self-testing mechanisms of the MUXTREE cell.

4.3.1 Testing the Memory Sub-System

In section 4.2.1 it was established that the memory element of embryonic cells is in fact a long shift-register whose content must remain unchanged during all the operational life of the array. This fact facilitates the task of testing the structure. The fault model assumed here is the well-known and commonly used "stuck-at" fault model. In this model all faults are of the stuck-at-0 and stuck-at-1 type [Lal96]. Under this assumption it is possible to test the integrity of the shift register during the configuration phase by appending 2 flip-flops and a gate at the end of the memory element, as shown in figure 4.7.

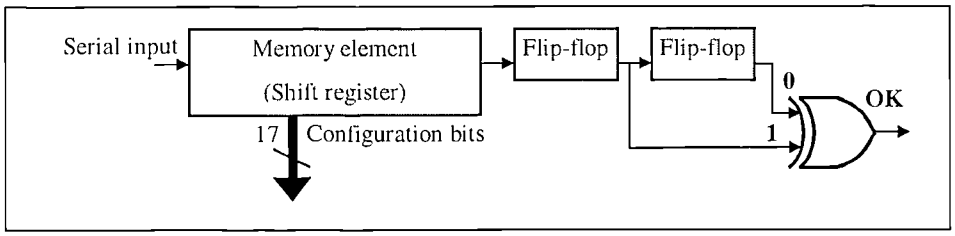


Figure 4.7 Self-test in memory element

If any of the bits in the memory element were stuck to a logic value, then the shift register would be filled with that value from that position onwards. The sequence 1,0 must be inserted at the beginning of the configuration frame of each column in order to set the diagnostic flip-flops to the adequate values. If one or more bits in the register were faulty, then both diagnosis bits would have the same value and the output of the gate would take the value 0, activating the reconfiguration process described in section 4.3.5.

The testing procedure just described detects memory faults only during configuration phase. There is no test procedure to verify the integrity of memory's content during the operational phase. To overcome this limitation, one alternative is to periodically re-load the genome. This solution implies a system stop while the new configuration bits are being downloaded. If the system is being evolved by means of a genetic strategy, then the genome would be continuously updated and the testing procedure proposed would be sufficient to thoroughly test the cell even during the operational phase.

4.3.2 Testing the Processing Element

Testing the processing element of the MUXTREE involves a very different set of problems from those involved in testing the memory [Tem97]. There are two facts that influence the design of the self-test strategy to be implemented:

- The selection element is too simple to allow a very sophisticated self-testing mechanism such as error-detecting codes [Lal85]. Such solution would imply additional logic that would certainly surpass the logic needed for the selector itself.
- The function of the self-test scheme is exclusively that of error-detection. Neither error masking nor error correction is needed; these functions are performed at a higher level.

Taking these facts into account, the test strategy that seems more appropriate is module redundancy; i.e. components are duplicated and the outputs of both elements checked for equality using an XOR gate. This technique signals a fault when the outputs of replicated units mismatch.

The components tested following this method are the five multiplexers and the flip-flop that make up the processing unit shown in figure 4.4. Figure 4.8 shows the generic implementation of this strategy.

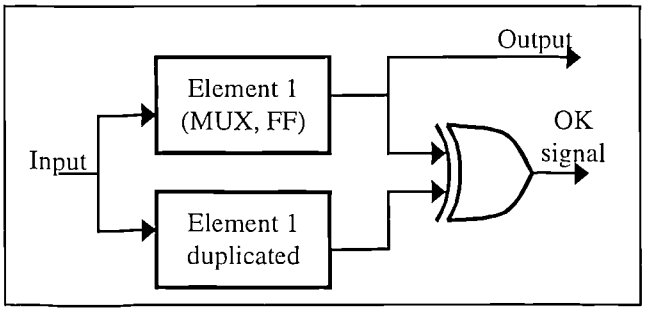


Figure 4.8 Testing of processing element. Generic diagram.

The six OK signals generated during the test of the multiplexers and the flip-flop are OR-ed together so that when any of these signals indicates a fault, a global OK signal is propagated to the reconfiguration logic in a higher level of the design.

4.3.3 Testing the Input/Output Router

It has been demonstrated that testing interconnection resources on FPGAs and VLSI systems is a complex and challenging task [Neg89, Tem97]. MUXTREE's reconfiguration strategy relies on its ability to re-route signals in order to avoid failing cells.

Since routing resources are just transmission paths for data ("wires"), the simplest way of testing them is by duplication or triplication (if error masking is desired) of all the lines, a very expensive approach in terms of the additional silicon area required. It was decided that any attempt to test the correct functioning of the I/O router would result in an unreasonable large and complex structure. Therefore no mechanism is incorporated and it is assumed that all the interconnection lines and logic were tested during manufacturing and they work

correctly throughout the lifetime of the MUXTREE. This assumption is in accordance with the present state of the art [Bel92, Gro94, Kun89, Tem97].

4.3.4 Cost of Built-In Self-Test Logic

Incorporating BIST logic in every cell of the embryonic array increases the silicon area needed for its implementation; nevertheless, for applications demanding high levels of reliability this cost is justifiable [Lee90].

Table 4.1 compares each one of the constituting elements of the embryonic cell with and without BIST. Comparisons are based on the number of modules, nets and equivalent nets reported by Viewlogic's Workview® synthesis tools. FPGA libraries from Actel® were used during schematic capture.

	Version without BIST logic		Version including BIST logic		
Component	**Modules**	**Equivalent Nets**	**Modules**	**Equivalent nets**	**Incr. (%)**
I/O Router	56	184	56	184	0.0
Processing element	50	219	142	528	141.1
Memory element	1470	3661	1482	3695	0.9
Complete cell	1591	4244	1706	4592	8.2

Table 4.1 Cost of incorporating BIST in the embryonic cell's design

The logic in Table 4.1 is calculated using the number of equivalent networks reported by the synthesiser. This figure has no meaning with regard to the physical implementation of the design because it does not take into account the routing, which is device-dependent; nevertheless, it is useful for comparing the relative complexity of circuits.

The relatively high increment in the complexity of the processing element due to BIST logic (141%) is absorbed by the low overhead incurred for testing the memory element (0.9%), which is by far the largest component in terms of silicon area required. It is clear from these results that further efforts in minimising the BIST logic for the processing element would have little impact on the overall size of the cell.

However, any effort in reducing the size of the memory block would improve cell reliability by reducing the number of logic gates needed to implement a complete cell.

4.3.5 Reconfiguration Strategies

When a fault is detected in one cell, a reconfiguration mechanism is triggered so that the array can reach a state in which it can still perform its functionality. Two strategies have been studied in this work: row-elimination and cell-elimination. The following sections explain the reconfiguration strategies. A formal analysis of embryonic arrays' reliability will be presented in chapter 5.

Row- / Column-elimination strategy

Row- and column-elimination are equivalent strategies. In the following discussion only row-elimination will be analysed, however, similar results apply for column-elimination.

In row elimination, the failing of one cell provokes the elimination of the corresponding row, which is substituted by the contiguous row to the north. Cells are logically shifted upwards until a spare row is reached and a new functional array is achieved. Figure 4.9 shows an example of row elimination in an array with one spare row.

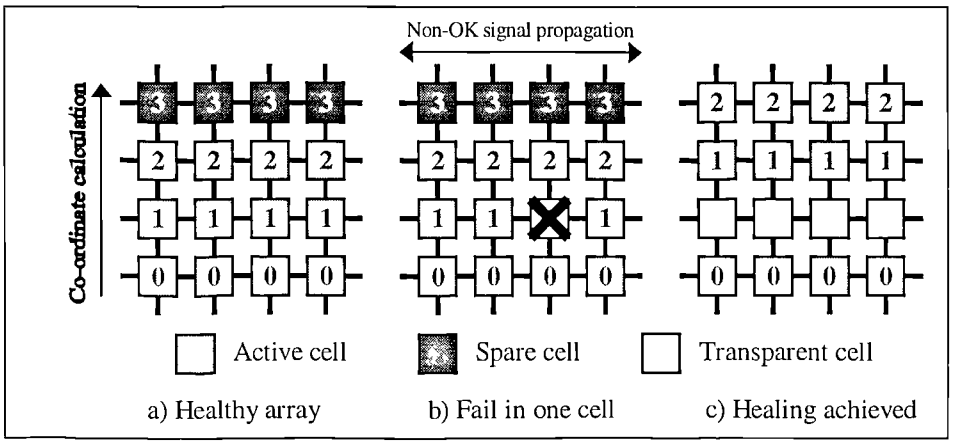


Figure 4.9 Reconfiguration by row-elimination in an embryonic array

When a fault is detected in any of the cells of the array, a non-OK signal is transmitted to all the cells in the row with the faulty cell. Cells receiving a non-OK signal become transparent for both the propagation of data and calculation of co-ordinates. Data signals are propagated from one side of cells to the other. Co-ordinates for cells above the row being eliminated are recalculated and new configuration registers are selected accordingly. No information is communicated around the array when reconfiguration takes place, only Boolean signals are passed between cells. The change in functionality of cells is achieved by simply using a different local memory location on each one.

It is important to note that when a failure occurs, the complete array loses one row. This strategy is far from being optimal with respect to the use of spare resources, but the short time needed to recover from a failure makes it attractive to implement real-time systems [All90]. A reliability analysis for this strategy is presented in section 5.4.1.

Cell-elimination strategy

By increasing the complexity of embryonic cells, it is possible to implement more sophisticated reconfiguration strategies. To achieve cell-elimination each embryonic cell is defined by two co-ordinates and donated with enough memory to contain the configuration registers of the entire array. This definition differs from the embryonics architecture described in previous sections where every cell is defined by one co-ordinate and stores only the configuration registers of the corresponding column.

In cell-elimination, spare cells replace faulty cells in two stages. First, spares located in the same row replace faulty cells. When the number of faulty cells in a row surpasses the number of spare cells, then the whole row is eliminated and cells are logically shifted upwards so that a spare row takes over the function of the failing one. Figure 4.10 shows an example of cell elimination in an array with one spare column and one spare row.

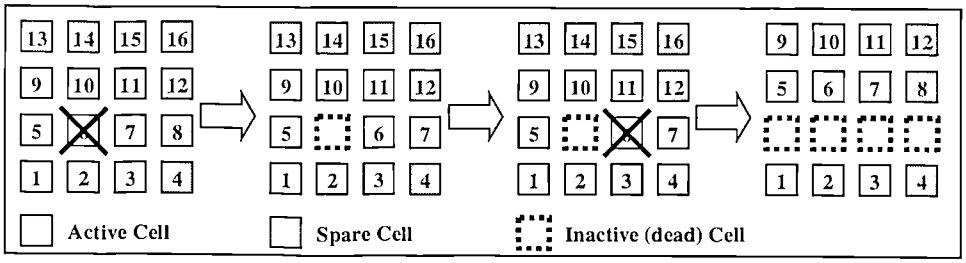


Figure 4.10 Fault-tolerance by cell elimination

The reliability analysis for the cell-elimination strategy is presented in section 5.4.2.

The basic embryonic cell can be designed so that it is possible to implement other reconfiguration strategies. However, more complex reconfiguration strategies imply increasing also the complexity on the cell, with the associated detriment in overall reliability.

An alternative approach called the MICTREE architecture has been followed by Mange and his research team [Man98a]. In the MICTREE architecture, a simplified version of the MUXTREE is used as the basic block to construct a hierarchical structure with embryonic characteristics at the highest of its levels. Details of the MICTREE architecture are given in section 5.4.3, along with its reliability analysis.

4.4 Application Examples

To verify the fault tolerance characteristics of embryonic arrays, three practical applications were implemented. The simplicity of the corresponding logic circuits allows a thorough inspection of embryonic arrays' internal behaviour. The logic circuits were mapped onto embryonic arrays following a methodology that is summarised in figure 4.11.

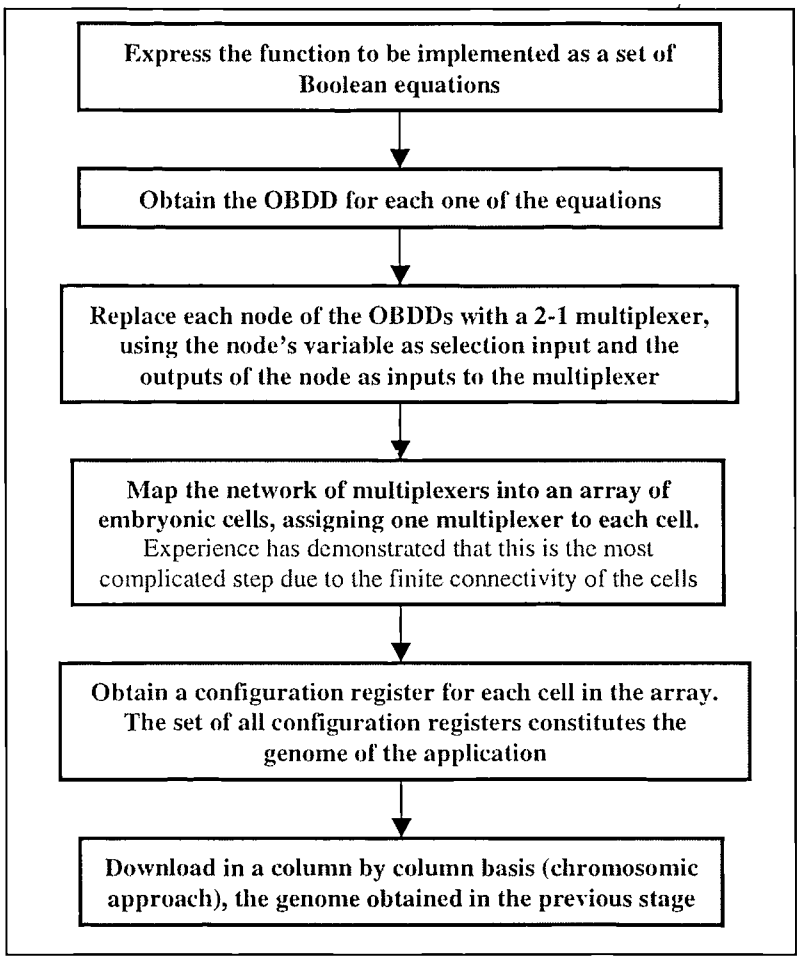


Figure 4.11 Methodology to map logic functions onto embryonic arrays.

4.4.1 Voter Circuit

The first example presents a combinational circuit that performs a voter function. Voters are used in fault-tolerant redundant systems to compare the output of replicated elements in order to detect and mask erroneous values [Bas95, Joh89]. In general, a voter receives n inputs and generates one output. The value at the output is the same as that received in at least $(n/2) + 1$ inputs. In a 3-input voter, the output is high or low if at least two of the inputs are high or low, respectively. The logic function that represents a 3-input voter is,

$$f(A,B,C)= AB + AC + BC$$

The voter function is also called the majority function because it delivers the value held in the majority of its inputs. Figures 4.12a and 4.12b show the OBDD for the voter function and the corresponding implementation using multiplexers. Figure 4.12c shows the voter implemented in a 3 × 2 embryonic array.

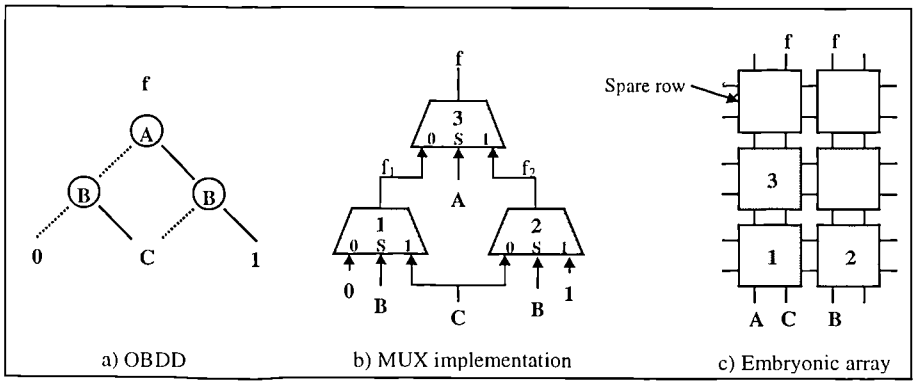


Figure 4.12 3-inputs Voter implemented in embryonic array

Figure 4.12 shows that three multiplexers are sufficient for a non-redundant implementation of the voter. To provide fault tolerance, one spare row is added to the embryonic array. If a fault is self-detected on one of the cells, then reconfiguration by row-elimination takes place; i.e. co-ordinates are shifted upwards until a spare row is reached.

One advantage of embryonic systems over conventional implementations is that outputs can be routed through several cells so that their value is presented in more than one output pin. The implementation shown in figure 4.12c is one of many possible mappings. Other ways for routing signals or distributing the function among cells are possible.

Figure 4.13 shows simulation results for the voter circuit. In this example, signals **OK1**, **OK2** and **OK3** indicate with logic zero that the corresponding physical cell in the array has failed, i.e. **OK** signals are related to cells in the array, not to the network of multiplexers.

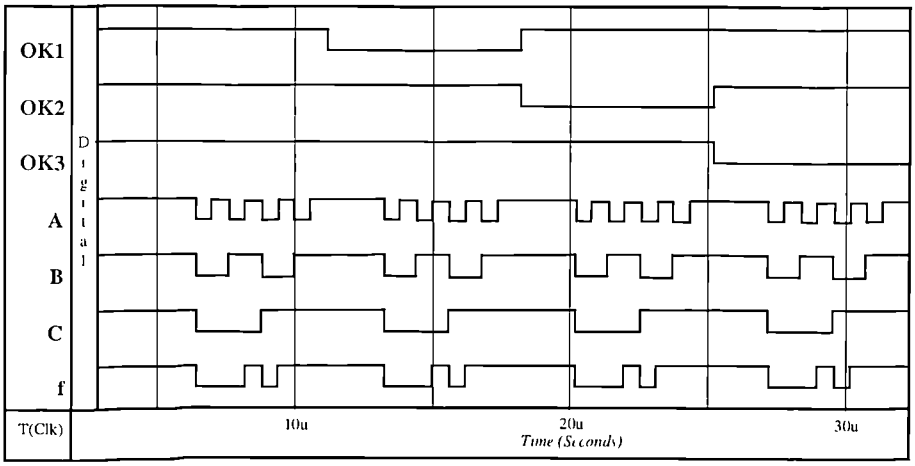


Figure 4. 13 Simulation of an embryonic array implementing a voter circuit

In figure 4.13 labels correspond with those of figure 4.12. **A**, **B** and **C** are the inputs to the voter and **f** is the output. All possible input combinations were tested and in every case the output of the system was consistent with the expected results when faults were simulated.

4.4.2 2-Bit Up-Down Counter

The second example is the implementation of a sequential circuit: a 2-bit up-down counter. The counter is driven by the system's clock **CLK**. Two outputs **A** and **B** maintain an increasing or decreasing binary count, in accordance to the logic value of input **U/D'**. If **U/D'** is high the counter increments, otherwise the counter decrements. Figure 4.14 shows the development of the counter from its transition table specification to its final implementation in a 4x4 embryonic array. Numbers in the cells of the array correspond to those of the multiplexers

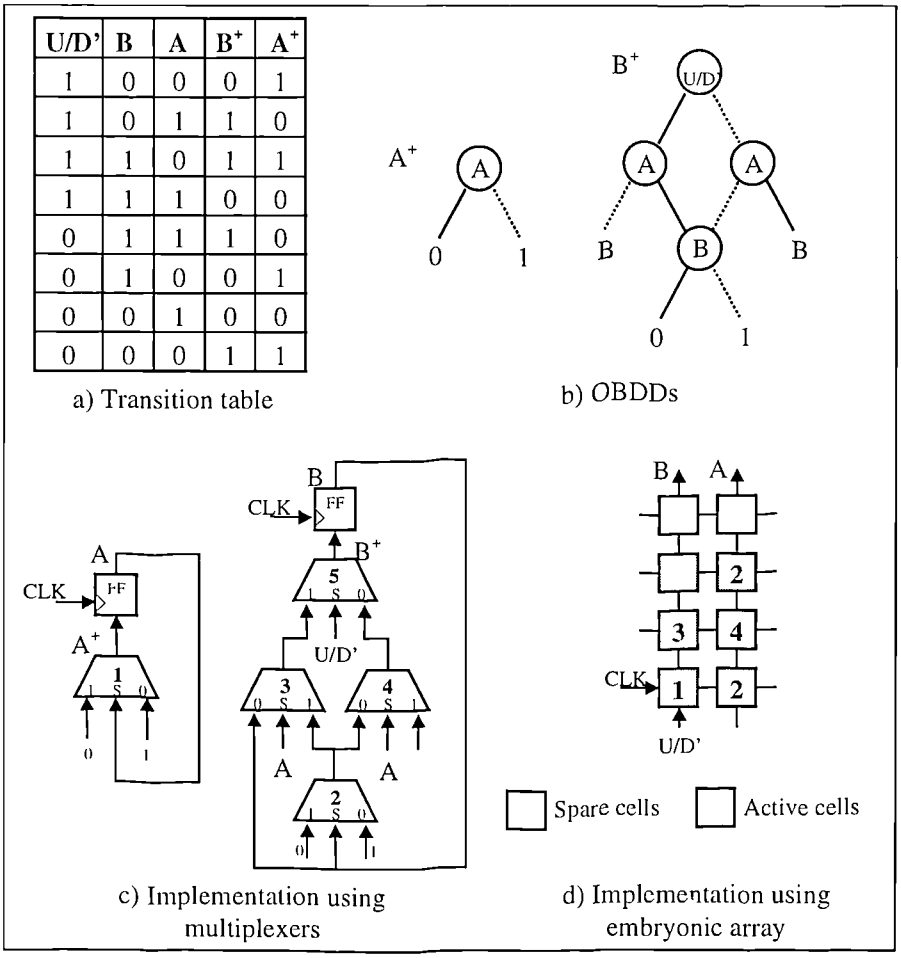


Figure 4.14 2-bits up-down counter

Figure 4.15 shows simulation results for the counter. Signal **U/D'** is the ascending/ descending control input. **OK4** is a simulation signal that injects a fault into the cell with multiplexer 4. Notice that when **OK4** goes to logic 0, there is a time interval on which the output of the counter is not reliable just before returning to normal behaviour. This is due to

the row-elimination process being carried out. The circuit settles after a number of clock-cycles that is proportional to the number of active rows involved in the reconfiguration. The process of co-ordinates re-calculation is carried out in the time-scale of nano-seconds; therefore spurious behaviour during this period cannot be seen in Figure 4.15.

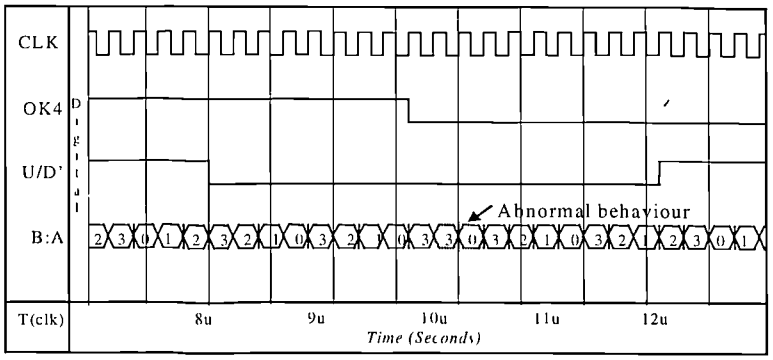


Figure 4.15 Simulation of 2-bit up-down counter implemented in an embryonic array

To prevent sequential designs from being "trapped" in spurious states during reconfiguration, it is responsibility of the designer to observe design rules to avoid such states [Eic65].

4.4.3 Programmable Frequency Divider

The design of a programmable frequency divider is presented next. This third example combines both combinational and sequential logic. A frequency divider receives a reference clock signal as input and generates a signal equal to the reference, divided by a constant factor specified by the user. A circuit that complies with the stated specification is composed of a 3-bit synchronous selector that latches either the division factor **n**, or the next state of a 3-bit down-counter, selected by a zero-detector. The signal used for reference is the system clock **F**. In this way, a 1-cycle wide pulse will be generated every **n** cycles of the reference frequency **F**. The output of the circuit is taken from the output of the zero-detector. It will be high during one cycle of **F** when the down counter reaches the 000 state. Figure 4.16 shows the circuit's block diagram.

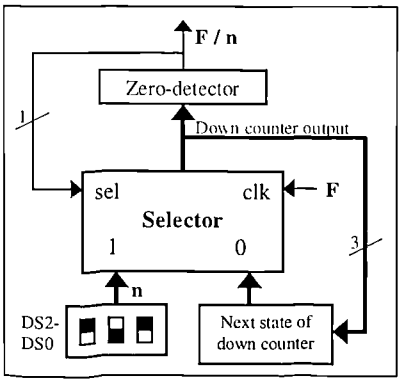


Figure 4.16 Programmable frequency divider

Figure 4.17 shows the OBDDs for the combinational down counter and zero-detector. **A**, **B**, **C** are the outputs of the 3-bit down counter, **C** being the most significant bit.

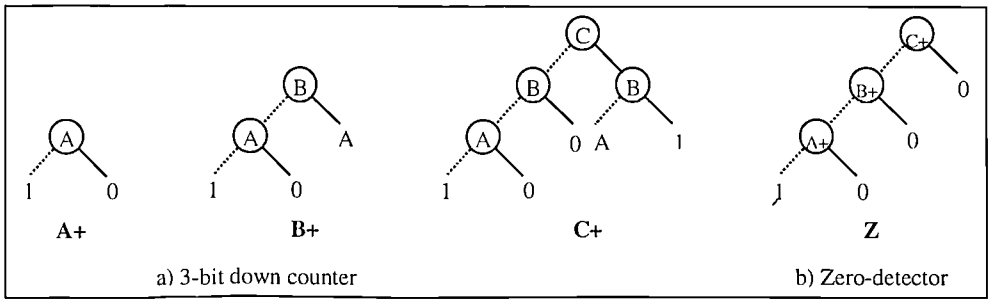


Figure 4.17 OBDDs for frequency divider

Figure 4.18 shows the hardware implementation of the OBDDs shown in Figure 4.17. Note that the OBDD for **A+** is repeated in the diagrams for **B+** and **C+**; hence the final circuit has been simplified. Multiplexers 1, 2 and 3 implement the selector block in figure 4.16.

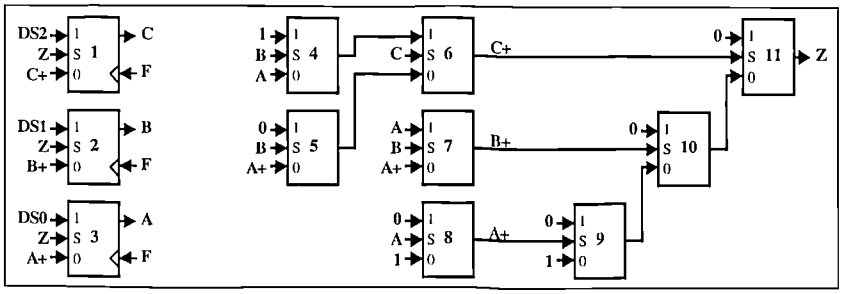


Figure 4.18 Hardware implementations of OBDDs for frequency divider

In figure 4.18 multiplexers 1, 2 and 3 operate in synchronous mode; i.e. their outputs are updated on the rising edge of **F**. External signals **DS2**, **DS1** and **DS0** set the value for **n**.

The circuits in figure 4.18 were mapped onto a 6x4 embryonic array. Figure 4.19 shows the final distribution of multiplexers. The numbers on each cell correspond with the numbers assigned to multiplexers in Figure 4.18.

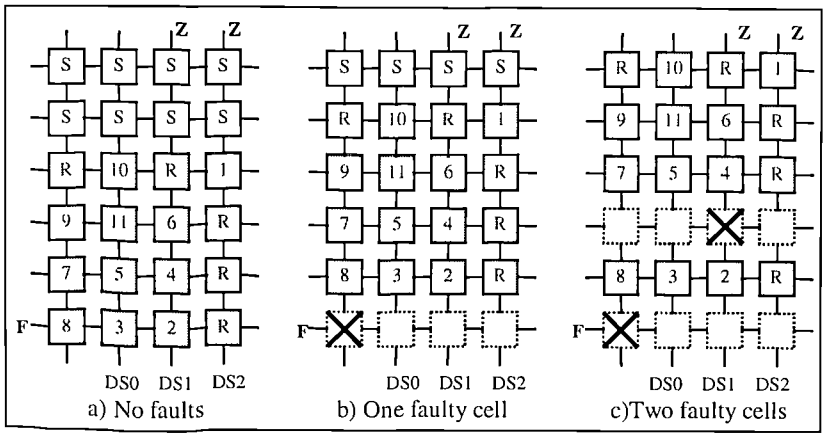


Figure 4.19 Frequency divider implemented in embryonic array

In figure 4.19, cells labelled **S** are spare cells, two rows for this example. Cells labelled **R** are routing cells. Routing cells are needed to propagate signal between non-neighbouring cells. The clock signal (**F**) is common to all the cells in the array.

Figure 4.20 shows typical simulation results obtained for the frequency divider. Labels correspond with those of figure 4.16. Although Figure 4.20 only shows reconfiguration in the divide-by-two region, thorough simulation was done in all other cases. **OK4** and **OK8** simulate faults in cells with multiplexers 4 and 8, as shown in Figures 4.19b and 4.19c.

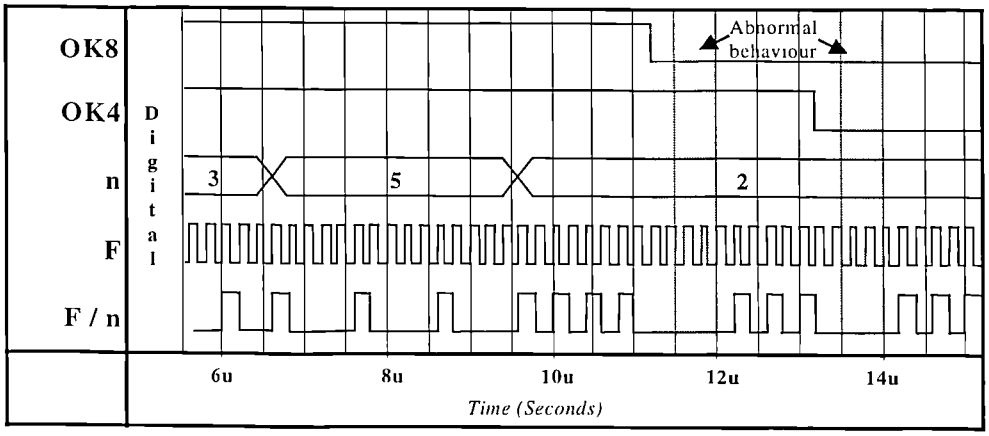


Figure 4.20 Functional simulation of frequency divider

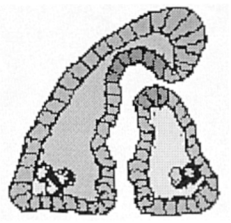
Notice that when **OK** signals go to logic 0, there is a time interval on which the output of the circuit is not reliable; however, after some clock-cycles, it returns to normal operation. This behaviour is due to the reconfiguration process being carried out and is considered the **healing period** of the system. The signals that propagate the internal state of the array (**OK** signals) could be used by an external mechanism to mask the effects of unreliable outputs during reconfiguration. Such mechanisms are beyond the scope of this thesis.

4.5 Summary

A detailed description of the MUXTREE embryonic architecture has been presented in this chapter. Simplicity in the number of components was sought after during the implementation of the blocks that make up the MUXTREE cell, working on the premise that a simple architecture is more reliable than a complex one. In the reliability analysis presented in chapter 5, the importance of simplicity will be thoroughly justified.

The examples that have been presented in this chapter have demonstrated that embryonic arrays possess fast fault-tolerant properties due to the uniqueness of their reconfiguration mechanisms. When a fault occurs only status signals are propagated, no configuration data is moved among cells.

Simulations of three examples have demonstrated that embryonic arrays achieve fast recovery from faults in combinational and sequential circuits. This characteristic is particularly useful for real-time applications where recovery time is a critical factor.



CHAPTER 5

RELIABILITY ANALYSIS OF THE EMBRYONICS ARCHITECTURE

This chapter provides a formal demonstration of embryonic array's fault tolerance. First, a review of the basics on reliability engineering is given in section 5.2. Section 5.3 presents reliability models of well-known system structures: series, parallel, k-out-of-m, and some combinations of them. In section 5.4, reliability models for the different reconfiguration strategies implemented in embryonics are obtained. In section 5.5 it is demonstrated that the models proposed can be used to compare reconfiguration strategies or different alternatives of a particular one.

5.1 Introduction

Embryonic arrays are proposed as a viable alternative to implement fault tolerance in processor arrays because of the autonomous and automatic reconfiguration mechanisms that are triggered when a fault is detected. This chapter presents the derivation of mathematical reliability models for different embryonic reconfiguration strategies. The models presented can be used both to evaluate different alternatives of a particular strategy, and to quantitatively compare different reconfiguration strategies.

Section 5.2 presents a succinct introduction to the subject of reliability engineering. Particular emphasis is given to the concepts of failure rate and Mean Time Between Failures. An important assumption taken throughout this work is that electronic components present a constant failure rate during their useful life. To support this assumption, a review of some widely used reliability-prediction procedures is presented.

Section 5.3 presents the basics of system reliability modelling. First, reliability models of systems whose elements are connected in series or parallel are derived. Next, it is demonstrated that by combining those simple models, reliability expressions for more complex systems can be obtained. Of particular importance is the *k-out-of-m* reliability model because it describes the behaviour of systems with spare units (hardware redundancy).

In section 5.4 the reliability models of the reconfiguration strategies explained in section 4.3.5 are presented. The reliability model for the MICTREE architecture proposed by Mange et al. [Man98] is also derived following the proposed methodology. The MICTREE architecture implements a reconfiguration strategy more complex than row- or cell-elimination. A discussion on the results drawn from the different system reliability models is presented in section 5.5.

5.2 Basic Definitions on Reliability

A fundamental problem in estimating reliability is whether a system will function according to its specification, in a given environment for a given period of time. This depends on factors such as the design of the system, the parts and components used, the complexity of the system, and the environment. Performance of a given system, under given conditions, for a given period of time can be considered a chance event, i.e. the outcome of the event is unknown until it has actually occurred. Hence it is natural to consider the reliability of a system as an unknown parameter which is defined to be the probability that the system will perform its required function under the specified conditions for a specified period of time.

According to the International Electrotechnical Commission (IEC), **reliability** has been defined as follows [IEC74]:

> **"Reliability is the capability of a product to perform its expected job under the specified conditions of use over an intended period of time"**

The formal study of reliability is a field on its own and a great deal of textbooks and periodical publications about the subject are printed every year. The following sections present a review of some important reliability concepts. It is not intended to be an exhaustive study, but aims to provide a theoretical background for the formulation of embryonics reliability models, later in this chapter.

5.2.1 Reliability and the Failure Rate

Consider the degradation of a sample of N identical components under stress conditions (e.g. temperature, humidity, vibration or radiation). Let $S(t)$ be the number of surviving components, i.e. the number of components still operating at time t after the beginning of the experiment, and $F(t)$ the number of components that have failed up to time t. Then the probability of survival of the components, also known as the reliability $R(t)$, is

$$R(t) = \frac{S(t)}{N}$$

The probability of failure of the components, also known as the unreliability $Q(t)$, is

$$Q(t) = \frac{F(t)}{N}$$

Since $S(t) + F(t) = N$, then $R(t) + Q(t) = 1$.

The **failure rate** $Z(t)$ is defined to be the number of failures per unit time compared with the number of surviving components:

$$Z(t) = \frac{1}{S(t)} \frac{dF(t)}{dt} \qquad (5.1)$$

Studies of electronic components show that under normal conditions the failure rate varies as indicated in figure 5.1. Because of its shape, figure 5.1 is commonly known as the **bathtub curve** [Mis92].

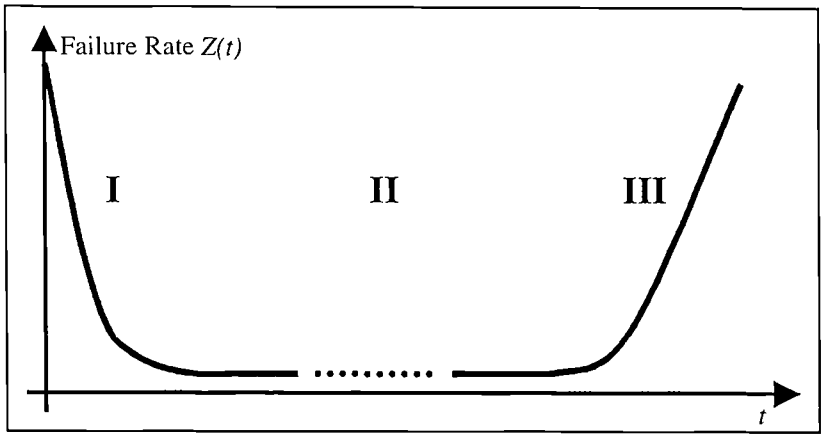


Figure 5.1 The bathtub curve

According to figure 5.1, in the life of any electronic component there is an initial period of high failure (region I). This is due to the fact that in any large collection of components there are usually some with defects and these fail immediately after they are put into operation.

For this reason, the first region is called the **burn-in** period of defective components. The middle phase is the **useful life** period (region II). In this region the failure rate is relatively constant and failures are random in time. The useful life is, under normal conditions, much longer than the other two phases. The final phase is the **wear out** period (region III), when the failure rate begins to increase rapidly with time. This is due to ageing of the components and accumulated stress.

In the useful life period the failure rate is constant, and therefore

$$Z(t) = \lambda \text{ (a constant)} \qquad (5.2)$$

With the previous nomenclature,

$$R(t) = \frac{S(t)}{N} = \frac{N - F(t)}{N} = 1 - \frac{F(t)}{N}$$

Therefore $$\frac{dR(t)}{dt} = -\frac{1}{N} \cdot \frac{dF(t)}{dt} \quad \text{or} \quad \frac{dF(t)}{dt} = -N\frac{dR(t)}{dt} \qquad (5.3)$$

Substituting equations (5.2) and (5.3) in equation (5.1)

$$\lambda = -\frac{N}{S(t)} \cdot \frac{dR(t)}{dt} = -\frac{1}{R(t)} \cdot \frac{dR(t)}{dt} \quad \text{since} \quad R(t) = \frac{S(t)}{N}$$

or $$\lambda \cdot dt = -\frac{dR(t)}{R(t)}$$

The above expression may be integrated giving

$$\lambda \int_0^t dt = -\int_1^{R(t)} \frac{dR(t)}{dt}$$

The limits of the integration are chosen in the following manner: *R(t)* is 1 at *t*= 0 and, by definition, at time *t* the reliability is *R(t)*. Integrating the last expression,

$$\lambda |t|_0^t = |\ln R(t)|_1^{R(t)}$$

$$\lambda t = -|\ln R(t) - \ln(1)|$$

$$-\lambda t = \ln R(t)$$

Therefore

$$\boxed{R(t) = e^{-\lambda t}} \qquad (5.4)$$

Expression 5.4 is generally known as the **exponential failure law**. The constant λ is usually expressed as failures per unit time, for example failures per hour or failures per 10^6 hours.

System failures, like component failures, can also be categorised into three regions of operation. The early system failures such as wiring errors, faulty interconnections and dry joints are normally eliminated by the manufacturer's test procedures. System failures occurring during the useful life period are supposed to occur because of component failures.

If a system contains k types of component, each with a failure rate λ_k, then the system failure rate λ_{ov}, is

$$\lambda_{ov} = \sum_{1}^{k} N_k \lambda_k \tag{5.5}$$

where there are N_k of each type of component.

5.2.2 Mean Time Between Failures

Reliability $R(t)$ gives different values for different operating times. Since the probability that a system will perform successfully depends upon the conditions under which it is operating and the time of operation, the reliability figure is not the ideal for practical use [Lal85]. More useful to the user is the average time a system will run between failures; this time is known as the **mean-time-between-failures** (**MTBF**). The MTBF of a system is usually expressed in hours and is given by,

$$MTBF = \int_{0}^{\infty} R_s(t)dt \tag{5.6}$$

According to expression (5.6) the MTBF is the area underneath the reliability curve $R(t)$ plotted versus t; this result is true for any failure distribution. For the exponential failure law,

$$MTBF = \int_{0}^{\infty} e^{-\lambda t} dt = -\frac{1}{\lambda} \left| e^{-\lambda t} \right|_{0}^{\infty} = \frac{1}{\lambda} \tag{5.7}$$

Expression (5.7) demonstrates that, assuming an exponential failure law, the MTBF of a system is the reciprocal of the failure rate. If λ is the number of failures per hour, the MTBF is expressed in hours. A graph of reliability against time is shown in figure 5.2.

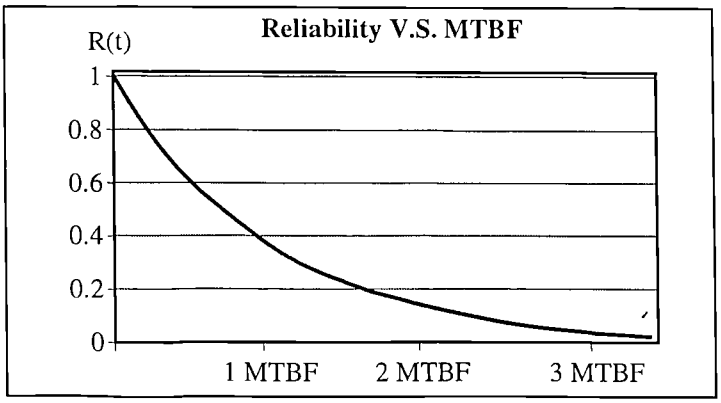


Figure 5.2 Reliability curve

Figure 5.2 shows that as time increases the reliability decreases and when t= MTBF, the reliability is only 36.8%. Thus a system with a MTBF of, for example, 100 hours, has only a 36.8% chance of running 100 hours without failure.

5.2.3 Reliability-prediction procedures

The term reliability-prediction has historically been used to denote the process of applying mathematical models and data for the purpose of estimating field-reliability of a system before empirical data are available for the system [Den98]. Reliability predictions are used in several important activities, for example:

- Feasibility evaluations,
- Comparing competing designs,
- Identifying potential reliability problems,
- Planning maintenance and logistic support strategies,
- Input to other studies such as life-cycle cost analysis or product selection.

Several reliability-prediction procedures have emerged as the field of reliability engineering developed into a mature subject. The six reliability-prediction procedures most widely used are [Bow92]:

1. United States Department of Defence Mil-Hdbk-217, Reliability Prediction of Electronic Equipment (MH-217) [Mil86].
2. British Telecom Handbook of Reliability Data for Components Used in Telecommunications Systems (BT-HRD4) [Bth84].
3. Bellcore Reliability Prediction Procedure for Electronic Equipment [Bell88].
4. Nippon Telegraph and Telephone Corporation Standard Reliability Table for Semiconductor Devices (NTT Procedure) [NTT85].
5. French National Centre for Telecommunications Studies Recueil de Donnees de Fiabilite du CNET (CNET Procedure) [CNE83].
6. Siemens Reliability and Quality Specification Failure Rate of Components [Sie86].

Although there is much discussion among reliability experts about which reliability-prediction procedure is the most accurate [Bow92, Lut90, Wat92], there is one common characteristic in the models proposed by all the procedures. All of them assume the exponential failure law to predict the reliability of a system; i.e. they predict reliability in the useful-life region of the bathtub curve (constant failure rate).

The constant failure rate (λ), is given by the device model. The device model is a function of parameters that describe its physical and operating characteristics, and the environment in which the device operates. Each reliability prediction procedure uses different environmental and quality factors to calculate λ. For example, in Mil-Hdbk-217, tables for use with a parts-count analysis give values of the generic failure rate, λ_G, for various microelectronic devices. Values for different environments are given, assuming nominal operating conditions and temperature for that environment.

Table 5.1 shows the values for λ and MTBF of a 64K DRAM memory according to the reliability prediction procedures mentioned above ([Bow92]).

Procedure	λ (Failures per 10^9 hours)	MTBF (years)
MIL-HDBK-217 (Parts count)	219	521
BT–HRD4	8	14,260
Bellcore RPP	140	815
NTT	138	827
CNET Procedure (simplified)	1950	59
Siemens	96	1188

Table 5.1 Predicted Failure-Rates and MTBF for a 64K DRAM

Table 5.1 shows that different reliability prediction procedures can compute very different values for the MTBF. These differences arise because the parameters chosen by each procedure to model a given environment and manufacturing conditions yield different values for λ. Much care must be taken when selecting a reliability prediction procedure to avoid misguiding results with either positive or negative connotations. It is also important to stress that although reliability predictions have been used successfully as a reliability engineering tool for five decades [Den98, Eva98], they are only one element of a well-structured reliability program and, to be effective, must be complemented by other elements.

For the purposes of this study, the relevant aspect of the reliability prediction procedures presented here is the fact that they all assume a constant failure rate for their analysis. The same will be assumed throughout the following sections.

5.3 System Reliability Modelling

Mathematical models are necessary to predict the behaviour of a system under certain specified conditions of use and operation. This section aims to develop reliability models for systems whose input-output conditions are specified and the conditions of use are known in advance. The following conditions will be assumed throughout:

1. Only catastrophic failures will be considered, i.e. failures are sudden and complete.
2. The states of all elements are statistically independent; i.e. the failure of one element does not affect the probability of failure of other elements.
3. Each element may be represented as a two-terminal device.
4. All the elements are initially operating.
5. Interconnections between elements are perfect.
6. The state of an element and of the system can be either good (operating) or bad (failure), i.e. there is not intermediate state.

Of the various models based on the functional interaction that two of more elements in a system can have, the following will be analysed: series, parallel, series-parallel, parallel-series and k-out-of-m.

5.3.1 Series model

A series model is the most common and the simplest reliability model. Such a model results if all the components in a system must operate successfully for the system success. Figure 5.3 shows the block diagram and the reliability logic diagram of a system whose elements are all connected in series. The reliability logic diagram shows all the possible interconnections between elements that make the system work. A combination is active if all its elements are working correctly at the same time. When more than one combination of elements satisfies the functionality of a system, at least one of the combinations must be active for the system to provide its service.

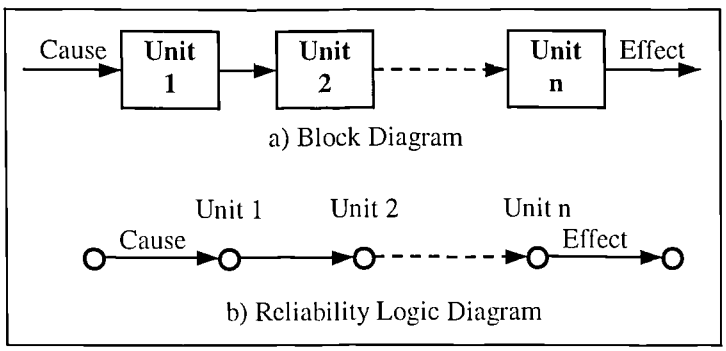


Figure 5.3 Block and reliability logic diagrams for the series model

Since *all* the units in a series system must function correctly, and given that the failures of the units are independent, then the reliability of the complete system would be given by the multiplication of the individual reliabilities,

$$R_s = \prod_{i=1}^{n} p_i \qquad (5.8)$$

Where p_i is the reliability of the *ith* component in the system.

Equation (5.8) shows that system reliability of the series configuration is much less than the reliability of any of its components. Hence, in order to design reliable systems, as few components connected in series as possible must be used.

If all units are identical and their reliabilities are given by equation (5.4) (exponential failure law for electronic components), then the reliability of a series system would be,

$$R_s(t) = \prod_{i=1}^{n} e^{-\lambda t} = e^{-n\lambda t} \qquad (5.9)$$

Using the definition for MTBF given in section 5.2.2, it is possible to determine the MTBF of a series system, whose reliability is given by expression (5.9),

$$MTBF_s = \int_0^{\infty} e^{-n\lambda t} dt = \frac{1}{n\lambda} \qquad (5.10)$$

It is interesting to note that, if the constituent elements have exponential failure distribution, the system failure distribution also remains exponential.

In conclusion, the following observations can be made for a series model:

- A series model provides a lower limit of system reliability. The reliability is worse than the worst element.
- Given that if any one of the units fails, the system fails, the dependency or independence of failures would make no difference in the series reliability model.

5.3.2 Parallel model

A parallel reliability model results if all the components in a system must fail for the system to fail. Success of any one component (or more) in the system implies system success. The parallel model is shown in figure 5.4.

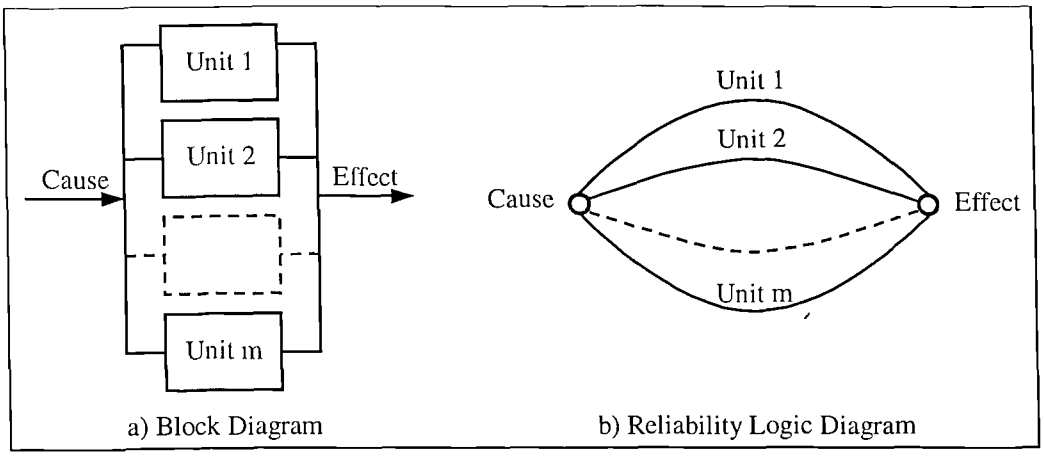


Figure 5.4 Block and reliability logic diagrams for the parallel model

The mathematical formula to represent the parallel model is developed through computation of probability of system failure. System failure occurs if all the system units fail. If all units were identical, the probability of system failure would be given by

$$Q_s = \prod_{i-1}^{m} q_i$$

Where q_i is the probability of failure for unit i.

Given that $R_s = 1\text{-}Q_s$, and $q_i = 1\text{-}p_i$, then

$$R_s = 1 - \prod_{i=1}^{m} q_i = 1 - \prod_{i-1}^{m} (1 - p_i)$$

For identical units following the exponential failure law,

$$R_s(t) = 1 - \prod_{i-1}^{m} \left(1 - e^{-\lambda t}\right) = 1 - \left(1 - e^{-\lambda t}\right)^m \qquad (5.11)$$

Expression (5.11) can be used to determine the MTBF of the parallel model. For example, in a parallel system with two units having failure rates λ_1 and λ_2 respectively, system reliability would be,

$$R_s(t) = 1 - \left(1 - e^{-\lambda_1 t}\right)\left(1 - e^{-\lambda_2 t}\right) = e^{-\lambda_1 t} + e^{-\lambda_2 t} - e^{-(\lambda_1 + \lambda_2)t}$$

And

$$MTBF_s = \int_0^{\infty} R_s(t) = \frac{1}{\lambda_1} + \frac{1}{\lambda_2} - \frac{1}{\lambda_1 + \lambda_2}$$

Similarly, system MTBF for 3-units parallel model is,

$$MTBF_s = \frac{1}{\lambda_1} + \frac{1}{\lambda_2} + \frac{1}{\lambda_3} - \frac{1}{\lambda_1 + \lambda_2} - \frac{1}{\lambda_1 + \lambda_3} - \frac{1}{\lambda_2 + \lambda_3} + \frac{1}{\lambda_1 + \lambda_2 + \lambda_3}$$

If failure rate is the same for all units ($\lambda_i = \lambda$) then,

$$MTBF_s = \frac{1}{\lambda} + \frac{1}{2\lambda} + \frac{1}{3\lambda} = \frac{1}{\lambda}\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3}\right) = \frac{1}{\lambda}\sum_{i=1}^{3}\frac{1}{i}$$

This result can be generalised to obtain the MTBF of a system with m units connected in parallel

$$MTBF_s = \frac{1}{\lambda}\sum_{i=1}^{m}\frac{1}{i} \qquad (5.12)$$

It is interesting to observe how the system MTBF improves as the number of parallel redundant units (m) increases. Table 5.2 provides the system MTBF of a parallel model for various values of m from 1 to 10.

m	System MTBF
1	1/λ
2	1.5/λ
3	1.833/λ
4	2.083/λ
5	2.283/λ
6	2.45/λ
7	2.593/λ
8	2.718/λ
9	2.829/λ
10	2.929/λ

Table 5.2 MTBF for parallel system

Figure 5.5 shows the graph of MTBF *versus* the number of units in a parallel system.

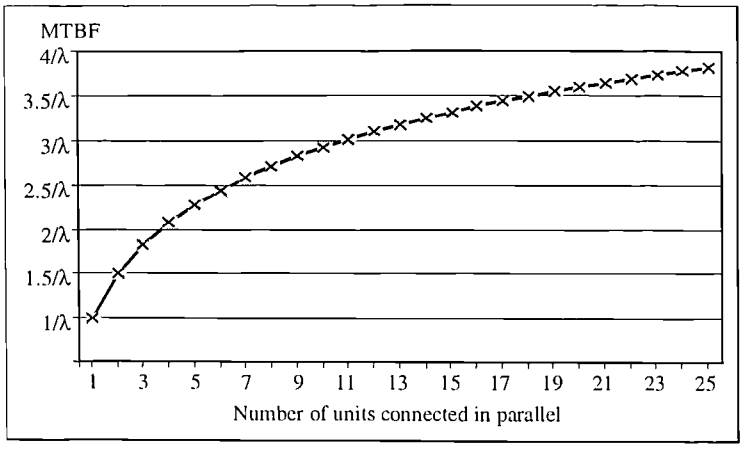


Figure 5.5 MTBF of parallel model

Table 5.2 and figure 5.5 show how the contribution to system reliability from each new unit that is added decreases as the total number of units increases. The graph in figure 5.5 shows that the MTBF of a parallel system with 25 units is less than 4 times longer than the MTBF of a single unit. In general, a parallel structure provides an upper bound for the reliability of a system consisting of m units; i.e. system reliability is better than the reliability of the best element.

5.3.3 Series-Parallel model

This structure consists of m_i elements in parallel to form a subsystem, and there are n such subsystems in series to constitute a complete system. For the system to work correctly, all its subsystems must function correctly. Figure 5.6 shows the block diagram of such structure.

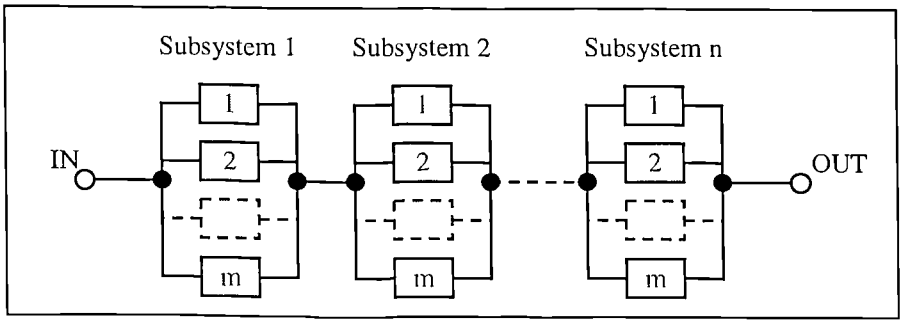


Figure 5.6 A series-parallel structure

Although the series-parallel structure is complex, it can be decomposed into, and analysed with, the basic series and parallel models. Assuming an exponential failure law for all its identical components, the reliability of the series-parallel model can be given by

$$R_s(t) = \prod_{i=1}^{n}\left(1-\prod_{j\ 1}^{m}\left(1-e^{-\lambda t}\right)\right) = \left(1-\left(1-e^{-\lambda t}\right)^m\right)^n \qquad (5.13)$$

The MTBF is obtained by integrating expression (5.13) [Mis92]. Integration is achieved by substituting for $x = 1-e^{-\lambda t}$ and transforming the variable from t to x.

$$MTBF = \frac{1}{\lambda}\int_0^1 \frac{\left(1-x^m\right)^n}{1-x}dx$$

But $(1-x^m)= (1-x)(1+x+x^2+\ldots+x^{m-1})$, therefore,

$$MTBF = \frac{1}{\lambda}\int_0^1 \frac{1-x^m}{1-x}\left(1-x^m\right)^{n-1}dx = \frac{1}{\lambda}\int_0^1 \left(1-x^m\right)^{n-1}\sum_{j-0}^{m\ 1} x^j dx = \frac{1}{\lambda}\sum_{j\ 0}^{m\ 1}\int_0^1 \left(1-x^m\right)^{n\ 1} x^j dx$$

This integral can be solved by using the transformation $x^m = y$,

$$MTBF = \frac{1}{m\lambda}\sum_{j\ 0}^{m\ 1}\int_{0}^{1} y^{\left(\frac{(j+1)}{m}\ 1\right)}(1-y)^{n-1}\,dy$$

By comparing the above expression with the Beta function defined as

$$B(\alpha,\beta) = \int_{0}^{1} x^{\alpha\ 1}(1-x)^{\beta-1}\,dx = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)} = \frac{(\alpha-1)!(\beta-1)!}{(\alpha+\beta-1)!}$$

It is possible to write MTBF expression as

$$MTBF = \frac{(n-1)!}{m\lambda}\sum_{j=0}^{m\ 1}\frac{\left\langle \frac{j+1}{m}-1\right\rangle!}{\left\langle \frac{j+1}{m}+n-1\right\rangle!} \qquad (5.14)$$

Where $\langle x\rangle$ means the largest integer not exceeding x.

For the case when m=1, equation (5.14) becomes equal to equation (5.10).

5.3.4 Parallel-Series model

This structure consists of n elements in series that form a chain or path, and there are m such paths in parallel to form a system. The system delivers its function as long as there is at least one path with all its elements working correctly. Figure 5.7 shows the block diagram of a parallel-series model.

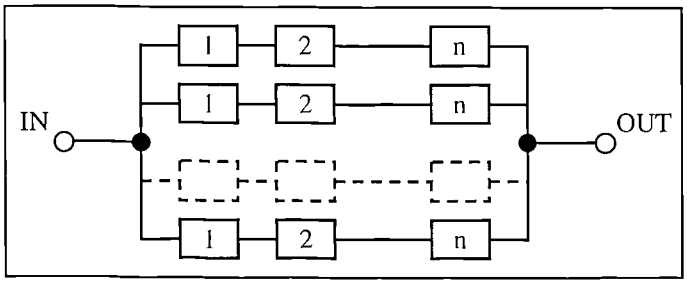


Figure 5.7 A parallel-series structure

Similarly to the series-parallel structure, the parallel-series can be analysed with the basic series and parallel models. Assuming that all the elements are identical and follow an exponential failure law, the reliability of the series-parallel model can be given by

$$R_s(t) = 1-\prod_{i\ 1}^{m}\left(1-\prod_{j=1}^{n}e^{-\lambda t}\right) = 1-\prod_{i-1}^{m}\left(1-e^{\ n\lambda t}\right) = 1-\left(1-e^{-n\lambda t}\right)^{m} \qquad (5.15)$$

The system MTBF would be $MTBF = \int_0^{\infty} 1-\left(1-e^{-n\lambda t}\right)^m dt$

The integral can be solved by substituting $y = 1-e^{-n\lambda t}$ as

$$MTBF = \frac{1}{n\lambda}\int_0^1 \frac{1-y^m}{1-y}dy = \frac{1}{n\lambda}\sum_{j=1}^{m}\frac{1}{j} \qquad (5.16)$$

Equation (5.16) can be interpreted as the MTBF of m parallel units, where each unit has a failure rate of $n\lambda$.

5.3.5 k-out-of-m model

In many situations, a system functions properly if any k out of m units function properly. The reliability logic diagram will have $\binom{m}{k}$ paths, and each path will have k elements in series. Figure 5.8 shows the block diagram and the reliability logic diagram of a system that performs correctly as long as three out of five of its units work correctly. The reliability logic diagram shows all the combinations of three units that make the system work. A combination is active if all its units are working correctly at the same time. At least one of the combinations must be active for the system to provide its service.

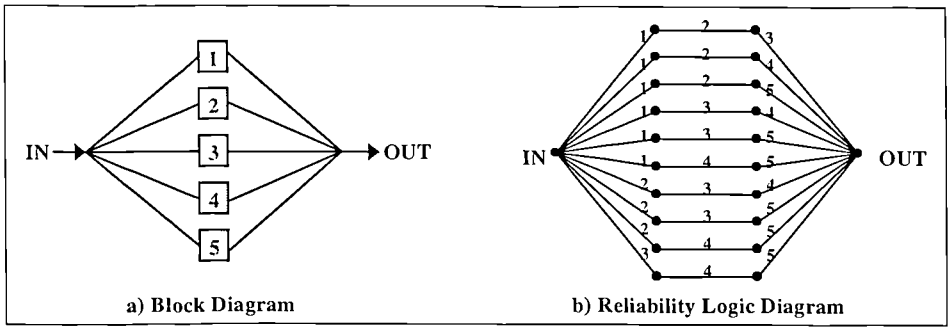


Figure 5.8 3-out-of-5 system

If all units are identical and p is the success probability of every unit, then the probability of exactly k units working correctly out of m is given by the binomial distribution,

$$B(k,m,p) = \binom{m}{k} p^k (1-p)^{m-k}$$

For the general case, the system remains functional as long as k, k+1... m-1 or m units function correctly. Therefore, the probability of system success is obtained by adding up the probability of all possible successful configurations,

$$R_s = \sum_{i=k}^{m} \binom{m}{i} p^i (1-p)^{m-i}$$

If all the units follow the exponential failure distribution, then

$$R_s(t) = \sum_{i=k}^{m} \binom{m}{i} e^{-i\lambda t} \left(1 - e^{-\lambda t}\right)^{m-i} \qquad (5.17)$$

Figure 5.9 shows the graphic representation of equation (5.17) for m=1024, λ=0.2 and different values of k.

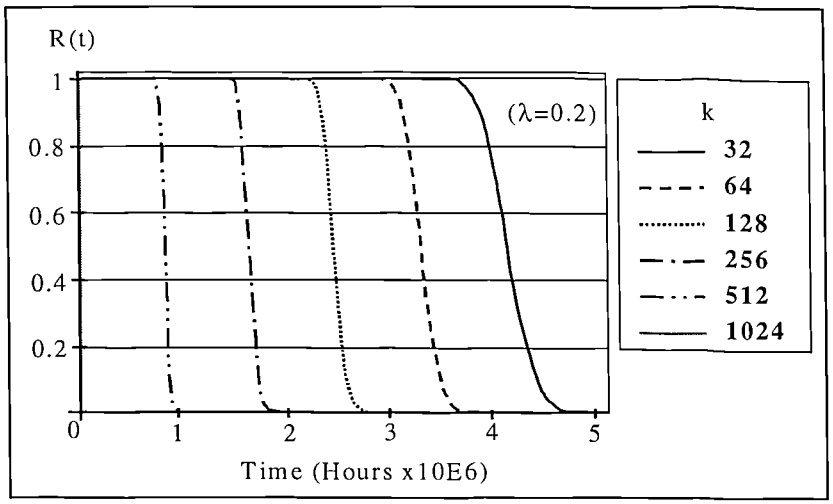


Figure 5. 9 Reliability of k-out-of-1024 system for different values of k

Figure 5.9 reveals that in a row with m cells an exponential increment in the number of active cells implies a linear decrement of reliability.

Figure 5.10 shows the behaviour of a 25-out-of-100 system for different failure rates.

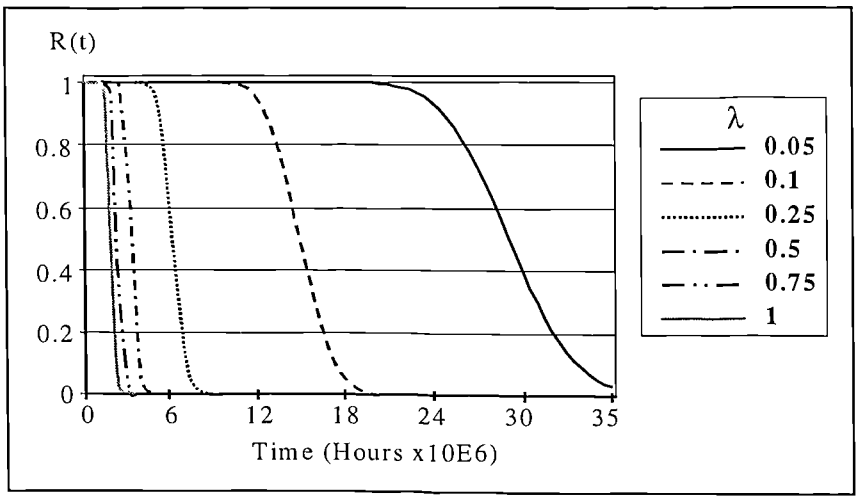


Figure 5.10 Reliability of 25-out-of-100 system for different failure rates

Figure 5.10 shows the high reliability of systems with small failure rates. Small improvements in the failure rate have an important positive impact in overall reliability, however decreasing the value of λ requires an improvement in the quality of the system's components and, in the majority of cases, the cost associated is too high.

The MTBF of a *k*-out-of-*m* system is obtained by recursion as follows [Mis92]:

Let *R(k,m)* and *MTBF(k,m)* denote the reliability and MTBF of a *k-out-of-m* system respectively. Then it can be observed that

$$R(k-1,m) = \binom{m}{k-1} e^{-\lambda t(k-1)} \left(1-e^{-\lambda t}\right)^{m-k+1} + R(k,m)$$

From to this equation, it is possible to write a recursive expression for the system MTBF as

$$MTBF(k-1,m) = \int_0^{\infty} \binom{m}{k-1} e^{-\lambda t(k-1)} \left(1-e^{-\lambda t}\right)^{m-k+1} dt + MTBF(k,m)$$

$$= \frac{1}{\lambda}\binom{m}{k-1}\sum_{j=0}^{m-k+1}\binom{m-k+1}{j}(-1)^j \frac{1}{k-1+j} + MTBF(k,m) = \frac{1}{\lambda(k-1)} + MTBF(k,m)$$

Using the identity that $\sum_{j=0}^{n}\binom{n}{j}(-1)^j \frac{1}{a+j} = \frac{n!(a-1)!}{(n+a)!}$ for $a \geq 1$, *MTBF(k,m)* can be solved recursively, starting with $MTBF(1,m) = \frac{1}{\lambda}\sum_{j=1}^{m}\frac{1}{j}$, as follows:

$$MTBF(2,m) = \frac{1}{\lambda}\sum_{j=1}^{m}\frac{1}{j} - \frac{1}{\lambda(2-1)} = \frac{1}{\lambda}\sum_{j=1}^{m}\frac{1}{j} - \frac{1}{\lambda} = \frac{1}{\lambda}\sum_{j=2}^{m}\frac{1}{j}$$

$$MTBF(3,m) = \frac{1}{\lambda}\sum_{j=1}^{m}\frac{1}{j} - \frac{1}{\lambda(3-1)} = \frac{1}{\lambda}\sum_{j=3}^{m}\frac{1}{j} \text{ and so on}$$

$$\therefore MTBF(k,m) = \frac{1}{\lambda}\sum_{j=k}^{m}\frac{1}{j} \qquad (5.18)$$

Table 5.3 provides MTBF of *k-out-of-m* systems for *m*=5 elements that are identical and have exponential failure distribution.

m→ / k↓	1	2	3	4	5
1	1/λ	3/2λ	11/6λ	25/12λ	137/60λ
2	×	1/2λ	5/6λ	13/12λ	77/60λ
3	×	×	1/3λ	7/12λ	47/60λ
4	×	×	×	1/4λ	27/60λ
5	×	×	×	×	1/5λ

Table 5.3 MTBF of k-out-of-m system with identical elements

Figure 5.11 shows a set of graphs of the MTBF of the *k-out-of-m* model for different values of k and m. Note that when k=1, the system is equivalent to a parallel model, whereas when k=m, the system is equivalent to a series model. In fact, the parallel and series models set the upper and lower limits of the set of graphs.

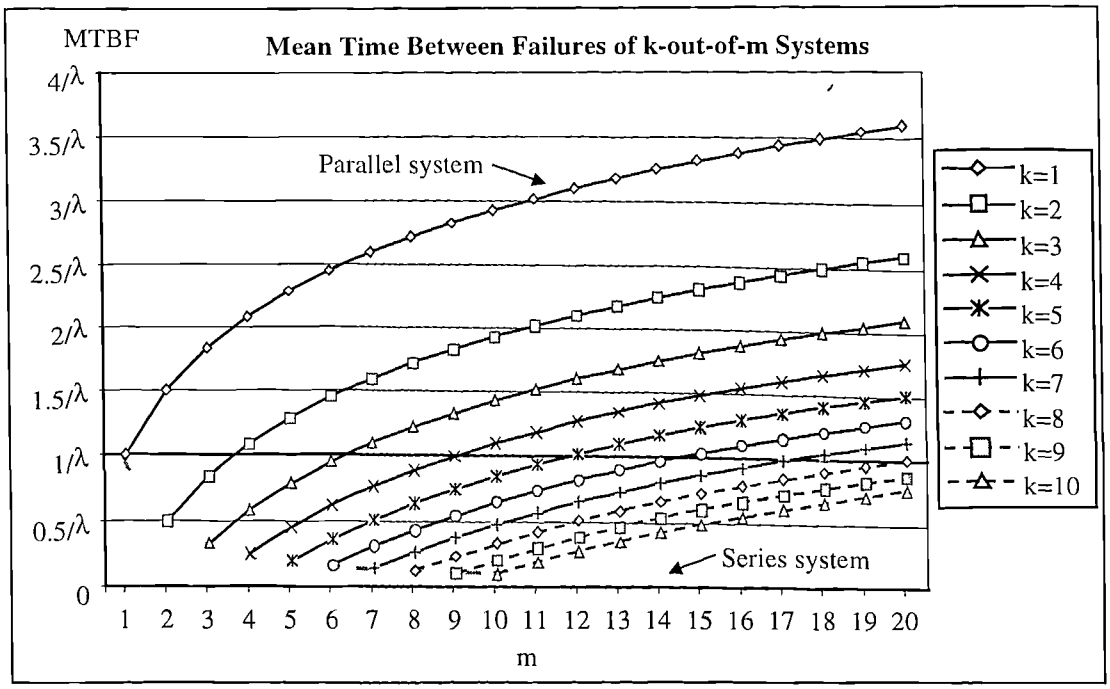


Figure 5.11 Graphs for the MTBF of k-out-of-m systems

In figure 5.11, all systems with a MTBF below the line corresponding to 1/λ have reliabilities worse than the reliability of a single unit. The graphs in figure 5.11 can be used to determine the minimum number of spare units needed to improve the reliability of a system above that point. For example, if a system requires 6 units to operate (k=6) then, according to the graph, a minimum of 9 spares are needed (m=15) to obtain a MTBF longer than 1/λ. Any value of m below 15 would negatively impact system reliability.

5.4 Reliability Models of Embryonics Reconfiguration Strategies

In order to propose a reliability model for the reconfiguration strategies of embryonic arrays, the reliability models presented in section 5.2 will be applied to analyse the reliability of two-dimensional arrays. To achieve fault tolerance embryonic arrays exploit the fact that integrated processor arrays have a fixed number of cells and in the majority of cases not all the cells are used. In embryonic arrays, those unused cells are used as spares.

In the following analysis, an array of size $n \times m$ will require at least a sub-array of size $r \times k$ working correctly in order to be considered in perfect working order. Figure 5.12 illustrates these assumptions.

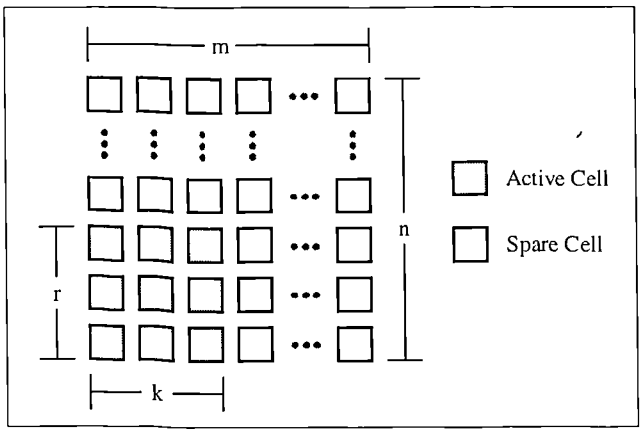


Figure 5.12 Cellular system with spares

In figure 5.12, active cells are the minimum number of cells needed to perform a required function. Spare cells are powered-up, but do not contribute to the normal operation of the system, they only become active when substituting faulty cells (i.e. hot sparing). Under this mode of operation the reliability of a spare cell is the same as that of any active cell; therefore, a failing spare cell can also trigger the reconfiguration mechanisms described in the following sections.

Cells in the array are the basic elements in the models to estimate reliability. According to embryonics fundamentals all cells are assumed identical; therefore, a constant failure rate λ is associated to all the cells (i.e. exponential failure law).

5.4.1 Row-elimination

Row- and column-elimination are equivalent cell-replacement strategies. In the following discussion only row-elimination will be analysed, however, similar results apply for column-elimination.

In row elimination, the failing of one cell provokes the elimination of the corresponding row, and cells are logically shifted upwards until a spare row is reached. After reconfiguration, the array continues delivering its function [Neg89]. Figure 5.13 shows an example of row elimination in an array with one spare row.

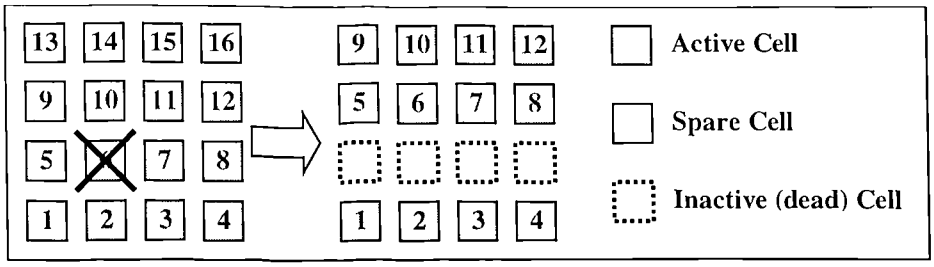


Figure 5.13 Fault tolerance by row-elimination

Even though this strategy eliminates many good cells when a fault occurs, the algorithm to carry it out is very simple and therefore, fast and easy to implement in hardware. In addition, figure 5.14 shows how as a square array becomes larger (more than 100 cells); the percentage of cells lost during reconfiguration decreases dramatically.

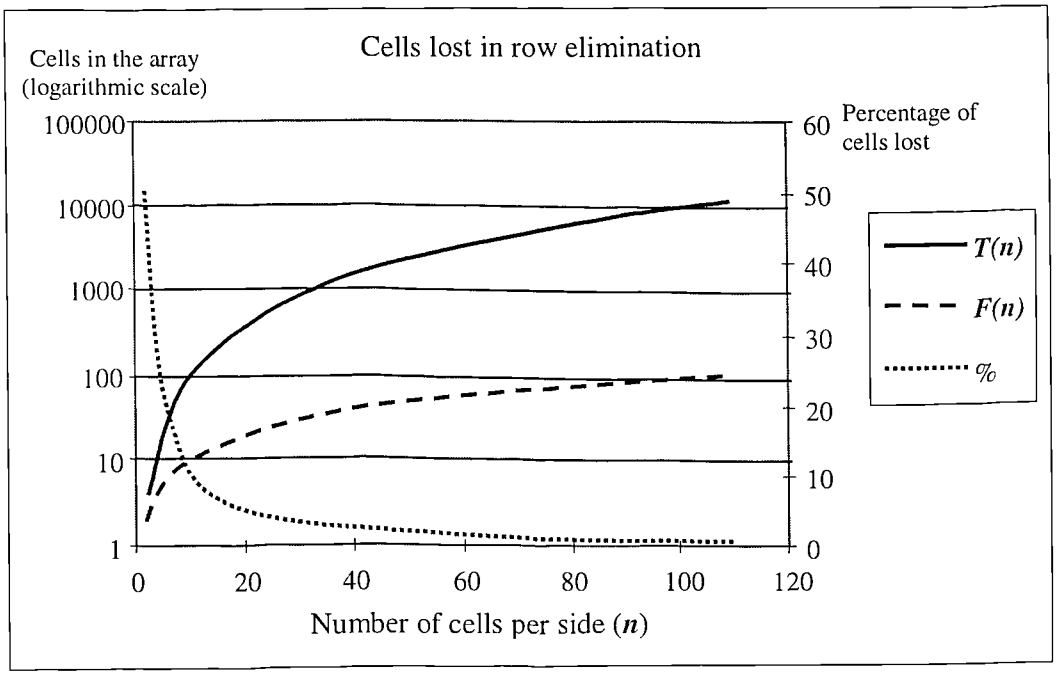


Figure 5.14 Percentage of cells lost during row-elimination

In figure 5.14:

n = Number of cells per side on a square array.

$T(n)$ = Total number of cells in the array = n^2

$F(n)$ = Number of cells that are eliminated when one cell fails = n

%= Percentage of cells lost during reconfiguration by row elimination = $\frac{F(n)}{T(n)} x100 = \frac{100}{n}$

Arrays with a large number of cells are particularly well suited for ALife applications where the emergent behaviours of multicellular systems are to be observed.

For the purpose of reliability analysis, cells in a row are connected in series; therefore, the reliability for a row $R_{rr}(t)$, would be given by the multiplication of the reliability distributions for all the cells in the corresponding row (equation 5.9),

$$R_{rr}(t) = \prod_{i=1}^{m} p_i(t) = \prod_{i=1}^{m} e^{-\lambda_R t} = e^{-m\lambda_R t} \qquad (5.19)$$

In (5.19), λ_R is the failure rate of an individual cell.

With the row's reliability determined, the array can be considered an r-out-of-n system, with r being the number of active rows needed for a particular application and n being the total number of rows in the array. Reliability of the whole array $R_{tr}(t)$, would be given by (5.17),

$$R_{tr}(t) = \sum_{j=r}^{n} \binom{n}{j} e^{-jm\lambda_R t} \left(1 - e^{-m\lambda_R t}\right)^{n-j} \qquad (5.20)$$

Figure 5.15 shows the graphs for equation (5.20) corresponding to an embryonic array of size $n \times m = 100 \times 25$, where r rows are needed to perform some function. The reliability of a 25-cell row with no redundancy is shown to allow a direct comparison against system reliability. Note how the shape of the graphs becomes steeper as the number of active rows increases.

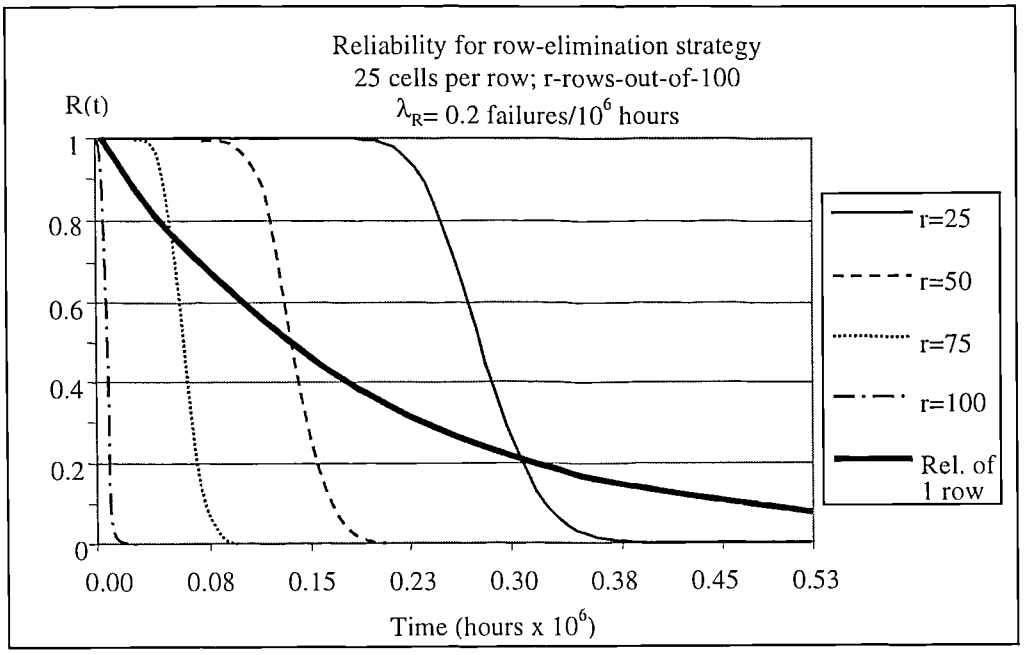


Figure 5.15 Reliability for row-elimination strategy

Figure 5.15 reveals that the reliability of the system is improved by the use of spare cells, and that long MTBF can only be achieved by using a large number of spare rows.

5.4.2 Cell-elimination

In cell-elimination, spare cells replace faulty cells in two stages. First, spares located in the same row replace faulty cells. When the number of faulty cells in a row surpasses the number of spare cells available, then the row-elimination strategy is adopted. Figure 5.16 shows cell elimination in an array with one spare cell per row and one spare row.

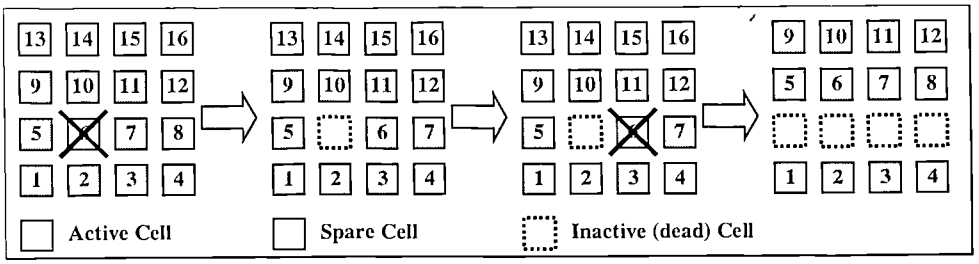


Figure 5.16 Fault-tolerance by cell elimination

During the first stage of reconfiguration each row of the arrays is itself a k-out-of-m system; therefore, the reliability for each row $R_{rc}(t)$, would be given by (5.17)

$$R_{rc}(t) = \sum_{i=k}^{m} \binom{m}{i} e^{-i\lambda_C t} (1 - e^{-\lambda_C t})^{m-i} \qquad (5.21)$$

Where λ_C is the failure rate of one individual cell in the array performing cell-elimination.

To analyse the second stage of reconfiguration, the array performing cell-elimination is considered a k-out-of-m system, where the basic element is one complete row. System reliability $R_{tc}(t)$, is obtained by recursively substituting the reliability of rows $R_{rc}(t)$ in (5.17).

$$R_{tc}(t) = \sum_{j=r}^{n} \binom{n}{j} R_{rc}(t)^j (1 - R_{rc}(t))^{n-j} \qquad (5.22)$$

Figure 5.17 shows the reliability of a system requiring r rows out of 100 to accomplish its function. Rows are 25-out-of-50 systems.

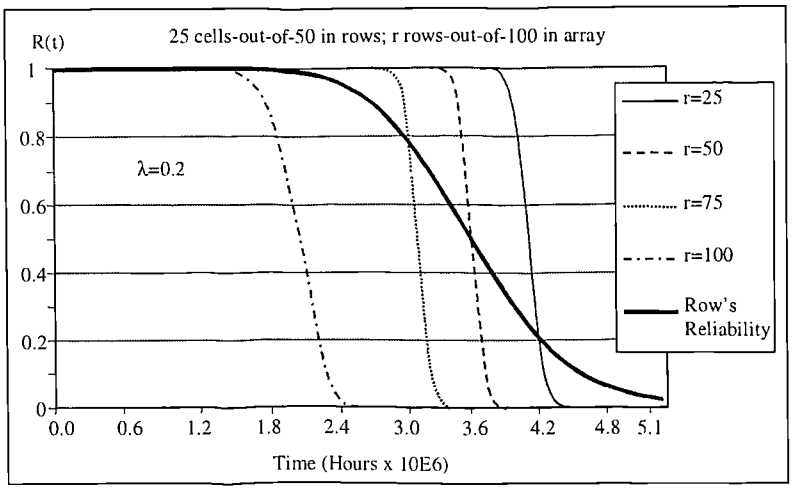


Figure 5.17 Reliability for cell-elimination strategy

Cell-elimination provides a very efficient use of spare cells, but the complexity of cells increases due to the extra logic needed to re-route data after reconfiguration. Therefore, the failure rates of the individual cells used in arrays performing row- and cell-elimination (λ_R and λ_C, respectively) should be different. In accordance with the reliability prediction procedures outlined in section 5.2.3, it is possible to assume the increase in λ_C linearly proportional to the extra resources (i.e. logic cells, gates or transistors) used to route data. An exact model that relates failure rate and the complexity of a cell is technology-dependant and, therefore, must be empirically estimated for each particular case.

5.4.3 Reliability of MICTREE architecture

Mange and his research group at the Federal Polytechnic of Lausanne in Switzerland have applied the MUXTREE cell to construct a hierarchical cellular architecture [Man98]. In this approach, a simplified version of the MUXTREE cell is called a ***molecule***. Molecules can perform the selection function, but they do not contain the configuration registers of their neighbours. Section 4.3.4 showed that memory is the biggest element of the embryonic cell (up to 90% of the logic is used to implement the memory block), therefore molecules are extremely simple and consequently have very low failure rates (small λ).

An array of molecules is used to construct cells called **MICTREE**. MICTREE cells have embryonic characteristics; i.e. they have self-diagnosis capabilities and are also able to replace their neighbours by changing their co-ordinates. To construct a cell, an array of MUXTREE molecules implements a Turing Machine that is able to execute a set of microprogrammed instructions. Calculation of co-ordinates and implementation of logic functions are achieved by executing a micro-program inside each cell. A copy of the micro-program (genome) is passed to each cell during a configuration phase immediately after power-up.

Summarising, a MICTREE organism is composed by a linear array of MICTREE cells where every cell is itself an array of MUXTREE molecules. Figure 5.18 shows the hierarchical architecture of MICTREE organisms.

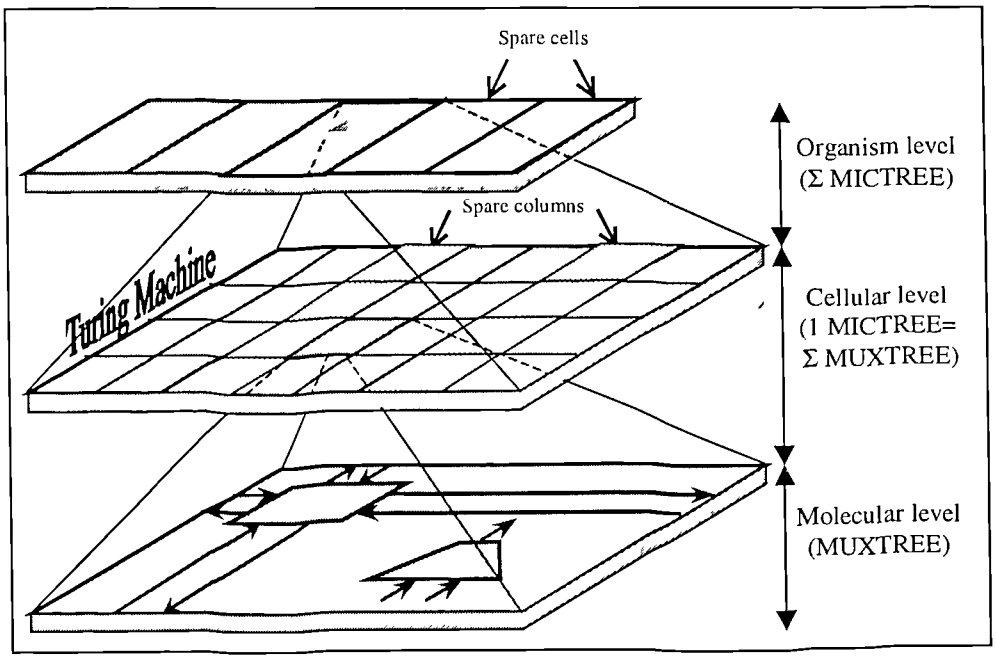


Figure 5.18 Hierarchical implementation of MICTREE array

Reliability of the MICTREE architecture

A generic MICTREE organism is made out of g MICTREE cells from which only f cells will perform the desired function. The remaining g - f cells are spares that replace faulty cells when a fault is detected. Figure 5.19 shows the structure of a MICTREE organism under these assumptions.

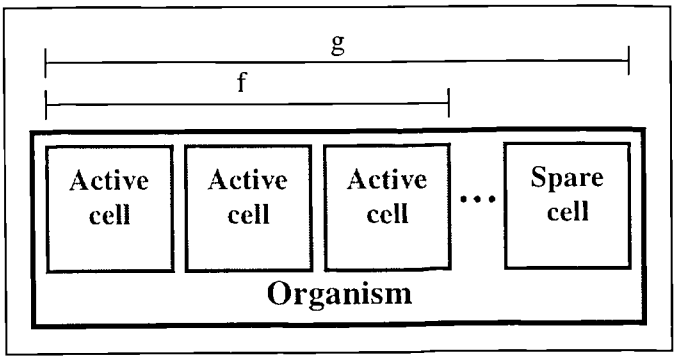


Figure 5.19 MICTREE organism

Every cell is itself a two-dimensional array of MUXTREE molecules. There are $x \times y$ molecules in a cell. A number of spare columns defined by the user can be inserted in a cell to provide a higher level of fault tolerance. Figure 5.20 shows the internal structure of a cell.

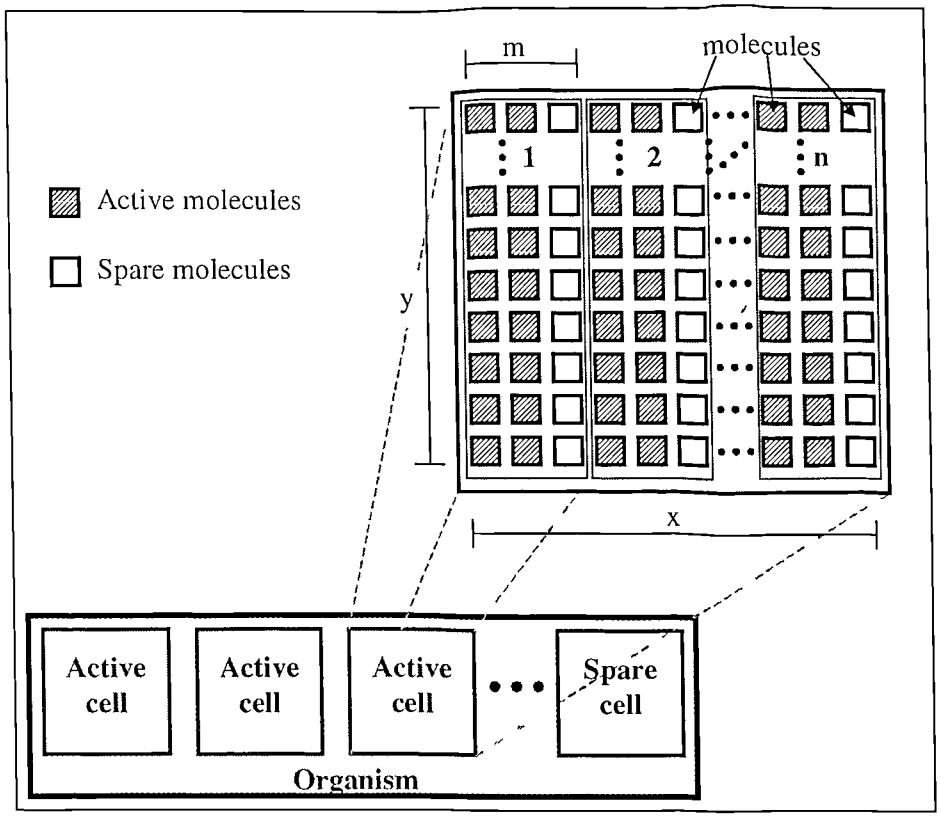


Figure 5.20 Internal structure of MICTREE cells

To analyse the reliability of the MICTREE architecture a bottom-up approach will be followed. Firstly, the reliability of one cell will be calculated; followed by the reliability of a complete organism. In the following analysis λ_M is the failure rate of a single MUXTREE molecule.

Reliability of cells

To analyse their reliability, cells are divided into **sub-arrays** of size $m \times y$ molecules, where every sub-array has one spare column. There are $n = x/m$ of such sub-arrays of molecules inside each cell, and all of them must provide their service successfully for the complete cell to be considered in working order. The reliability of one sub-array is given by the reliability of a set of m molecules organised as an $(m\text{-}1)$-out-of-m system, repeated y times. The reliability of one molecule is assumed to follow the exponential law given by $R(t) = e^{-\lambda_M t}$

Therefore, the reliability of one sub-array would be,

$$R_{sa}(t) = \left[\sum_{i=m-1}^{m} \binom{m}{i} e^{-i\lambda_M t} \left(1 - e^{-\lambda_M t}\right)^{m-i} \right]^y = \left[e^{-m\lambda_M t} \left(m e^{\lambda_M t} - m + 1 \right) \right]^y \quad (5.23)$$

In equation (5.23), λ_M is the failure rate of one molecule.

The reliability of one cell can be expressed as the series connection of n sub-arrays. Therefore, the reliability of one cell with n sub-arrays would be given by,

$$R_{cell}(t) = \left[e^{-m\lambda_M t}\left(me^{\lambda_M t} - m + 1\right)\right]^{xy} \qquad (5.24)$$

Figure 5.21 shows the graphic representation of equation (5.24) fór arrays with different values in the parameter x (width of the array). The following conditions have been assumed:

- Array's height (y) is 30 molecules
- Failure rate (λ_M) is 0.01 failures / 10^6 hours.
- Number of molecules per block (m) is 4

For the purpose of comparison, the reliability of a 30 × 50 molecules array with no spare columns is also shown in figure 5.21. The benefits of adding redundancy are evident.

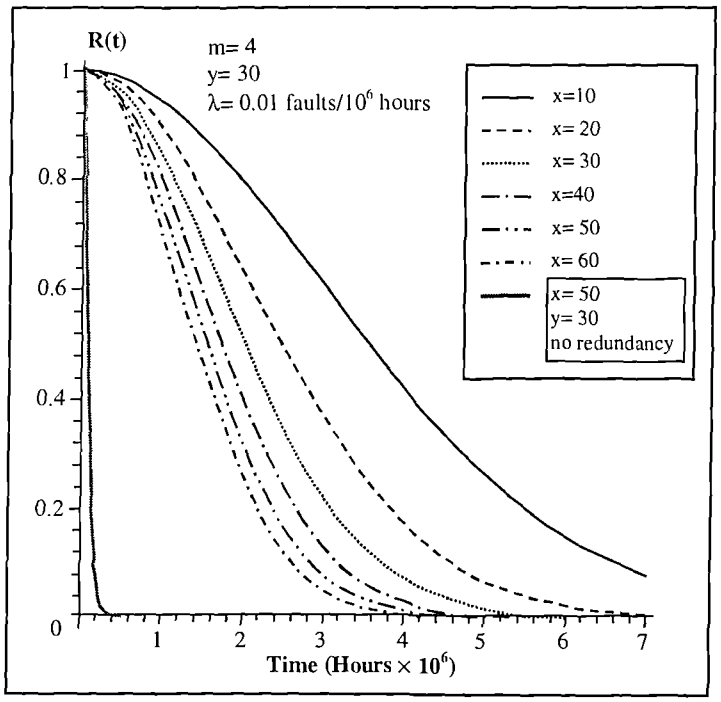


Figure 5.21 MICTREE cell's reliability for different array's widths

Figure 5.22 is a 3-D reliability graph of a cell with 30 rows (y) and 50 columns (x). The graphs for different values of m are shown. As before, m is the number of active columns between spare columns plus one, and the failure rate is set to 0.01 faults/10^6 hours.

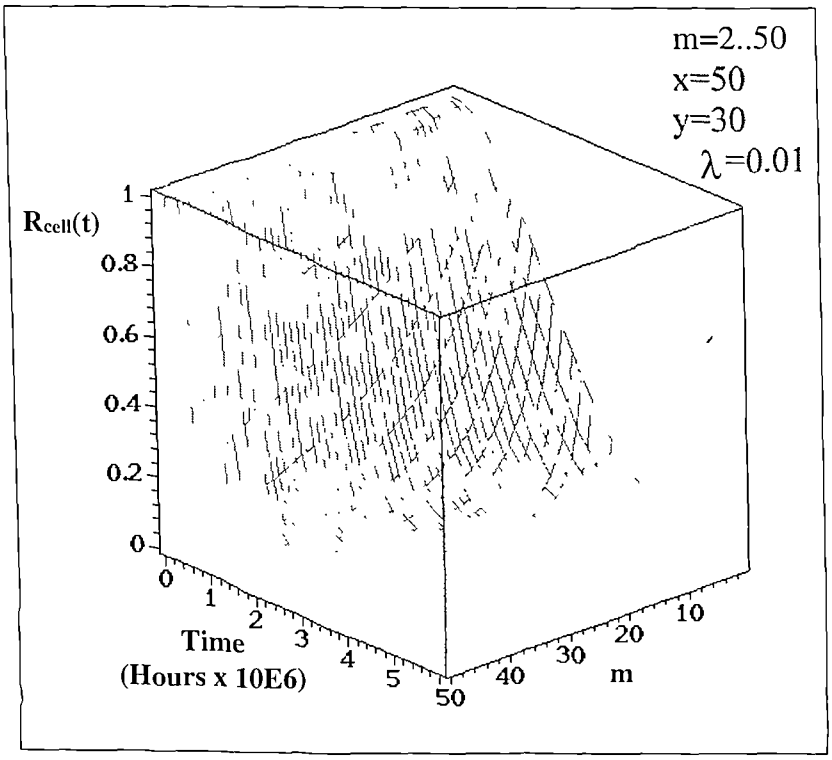


Figure 5.22 MICTREE cell's reliability for different values of ***m***

Figure 5.22 shows that the larger the number of spare columns in the array, the longer the cell will function correctly. It also shows that the relative improvement in reliability with respect to the number of spare cells increases as the number of spare cells increases. For example, the improvement in reliability when passing from 6 to 7 spare columns is bigger than the improvement when passing from 2 to 3 spare columns.

Reliability of organisms

For the purpose of reliability analysis, the MUXTREE organism is a system with g cells from which, f cells must function correctly in order to consider the organism in working order. Therefore, the organism's reliability can be modelled as an f-out-of-g system where the reliability of the constituent elements is given by the reliability of a MICTREE cell (equation 5.24) instead of the exponential law. Substituting (5.24) in (5.17) yields,

$$R_{org}(t) = \sum_{j=f}^{g} \binom{g}{j} R_{cell}(t)^{j} \left(1 - R_{cell}(t)\right)^{g-j}$$

$$= \sum_{j=f}^{g} \binom{g}{j} \left[e^{-m\lambda_M t}\left(me^{\lambda_M t} - m + 1\right)\right]^{jm} \left(1 - \left[e^{-m\lambda_M t}\left(me^{\lambda_M t} - m + 1\right)\right]^{m}\right)^{g-j} \qquad (5.25)$$

Parameters in (5.25) correspond to those shown in figures 5.19 and 5.20. Figure 5.23 shows the graphical representation of (5.25) for different number of active cells (f) in a 48-cells

organism. Each cell is assumed to be of size 30 rows (y) × 50 columns (x), with one spare column every two active columns (m = 3).

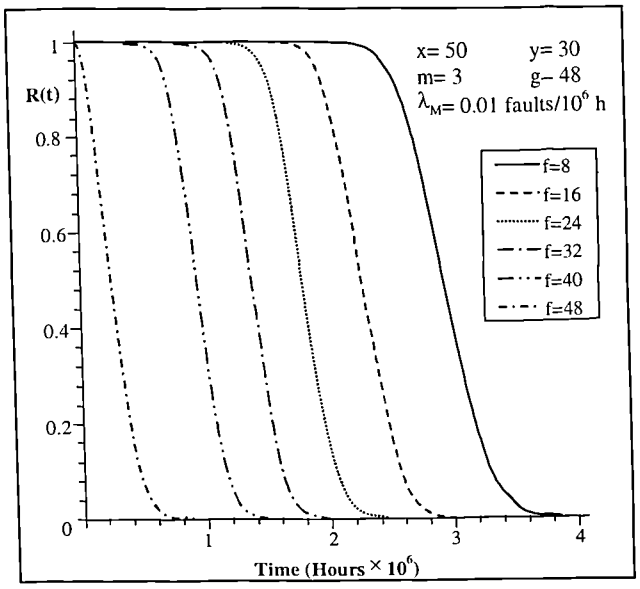


Figure 5.23 Reliability of 48-cells MICTREE organism for different number of active cells

In figure 5.23, the case when f = 48 represents an organism with no spare cells.

Figure 5.24 shows in detail the improvement of reliability in a 48-cells organism when adding eight spare cells one by one.

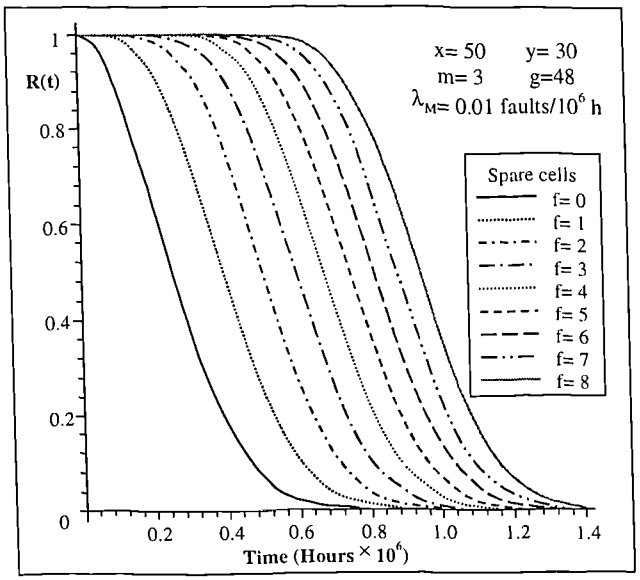


Figure 5.24 Reliability of MICTREE organism with different number of spare cells

Figure 5.24 reveals how the relative improvement in system reliability decreases as the number of spare cells increases. This is because spare cells have the same probability of failing as any other active cell.

Figure 5.25 shows the reliability of an organism with 40 active cells and 8 spare cells. Each graph corresponds to different number of spare columns at the cellular level. Cells are of size 30 × 50 molecules and the failure rate of each molecule is 0.01 failures/10^6 hours.

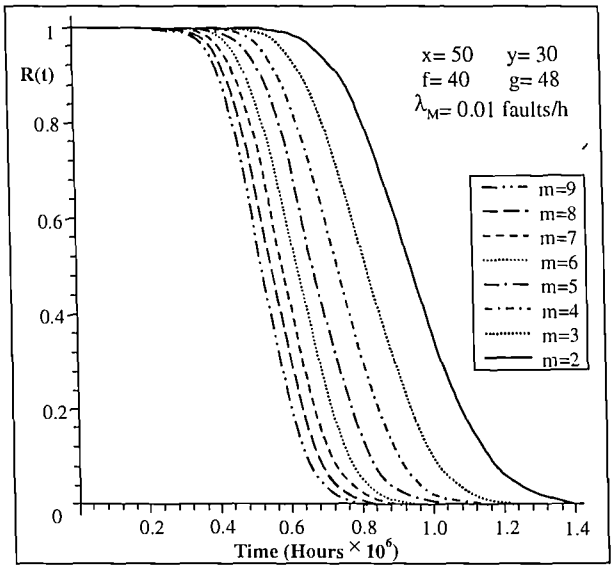


Figure 5.25 Reliability of MICTREE organism for different number of spare columns

Figure 5.25 shows that the relative improvement in cell's reliability increases as the number of spare columns increases. This is because the number of columns inside the cell remains constant and cell's reliability is mainly determined by the number of spare columns.

Figure 5.26 shows system reliability curves for a MICTREE organism with the following characteristics: 30 molecules × 50 molecules within cells, one spare column every two active columns (m= 3) in cells, molecule's failure rate = 0.01 failures/10^6 hours. At organism level the ratio of active cells over the total number of cells is kept constant, *i.e.* $f / g = 0.667$.

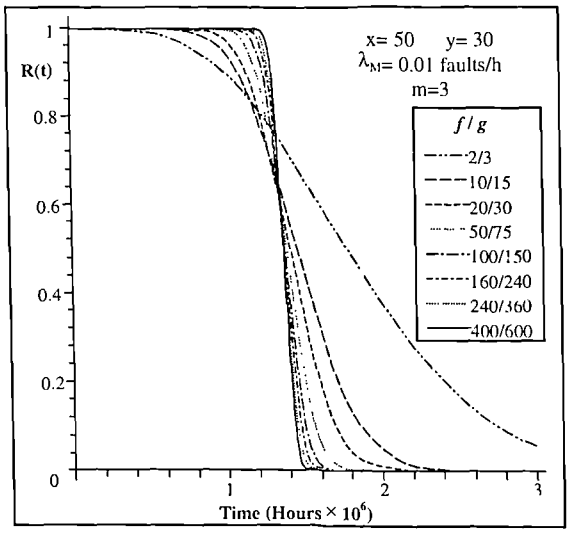


Figure 5.26 Reliability of MICTREE organism with different number of cells

Figure 5.26 shows that for a given active/total cells ratio the reliability curve becomes steeper as the total number of cells in the organism increases. The reliability graph of organisms with large number of cells approximates a step-shaped curve. In these cases, the Mean Time Between Failures (**MTBF**) of the system could be approximated to the time at which the reliability graph makes the transition from one to zero.

5.5 Discussion

It is clear that adding spares in parallel to a system will improve its reliability and its MTBF. A natural strategy to improve system's reliability would be to incorporate as many spares as possible; nevertheless, in some cases, the cost of spares can be very high. Therefore a tool that allows the quantitative comparison of different alternatives is needed. The reliability models presented in this chapter can auxiliate this decision-making process.

In the case of embryonics, graphs of the reliability expressions obtained for the reconfiguration strategies allow direct comparisons between different architectures. For example, consider a fixed array of size $n \times m = 50 \times 50$ MUXTREE cells (see figure 5.12). In which, the following architectures need to be compared:

1. An array performing row-elimination, i.e. no spares on rows.
2. An array performing cell-ellimination.

In order to make the comparison, the array's reliability for different number of spare rows is shown in figure 5.27

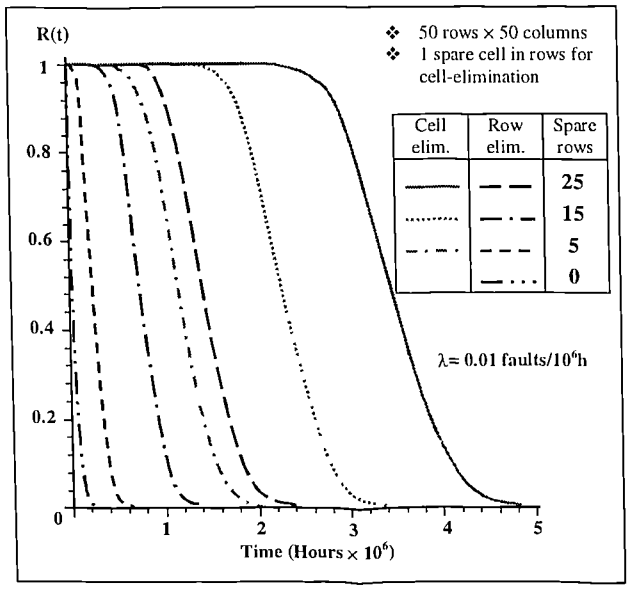


Figure 5.27 Comparison between row- and cell-elimination in a 50x50 embryonic array

Figure 5.27 shows that the improvement in reliability for adding a single spare cell per row in the cell-elimination strategy is remarkable (dark lines). It is interesting to observe that the graphs for row-elimination with 25 spare rows and cell-elimination with 5 spare rows are almost equivalent. Nevertheless, the array using cell-elimination uses 295 spare cells, whereas the array using row elimination uses 1250 spare cells out of 2500; over 4 times the number of cells used in cell-elimination.

Figure 5.28 shows, for the array used in the previous example, a comparison between an array using row-elimination and several alternatives using cell-elimination, each one with different number of active cells per row (k). In all the arrays, 10 spare rows per array have been considered (r=40). The graph for the reliability of the array without redundancy, i.e. 2500 cells connected in series, is also shown to allow a complete comparison of alternatives.

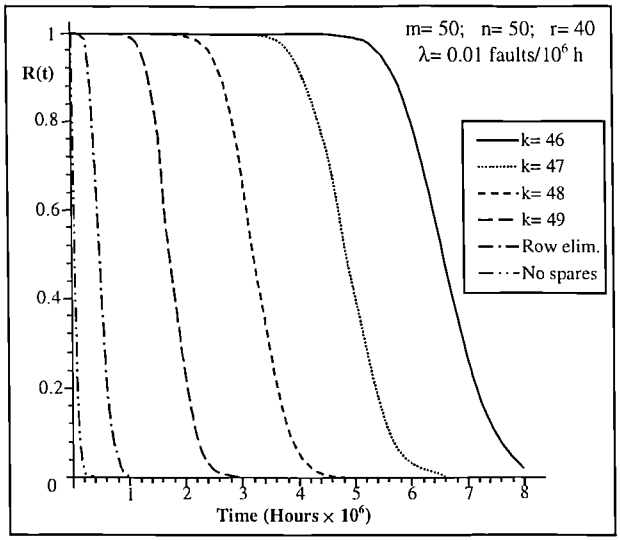


Figure 5.28 Reliability graphs for arrays with row- and cell-elimination

In figure 5.28, every graph of the cell-elimination strategy implies an increment of 50 spare cells in the array. Nevertheless, the number of cells in the array remains constant on every alternative, therefore, instead of considering an increment in the number of spare cells, a decrease in the number of active cells within the array must be assumed. According to this statement, the array using row-elimination will have 2000 active cells and 500 spare cells (10 spare rows of 50 cells each), whereas the arrays with one, two, three and four spare cells per row will have 1960, 1920, 1880 and 1840 active cells out of 2500, respectively. The number of active cells is given by

$$Active\_cells = r \times k \qquad (5.25)$$

To calculate reliability, the same failure rate for both row- and cell-elimination has been assumed in the preceding example. Nevertheless, in a real implementation, the cells on both arrays might not be equivalent in complexity. For example, the extra logic needed to perform cell-elimination will have a negative impact on the cell's failure rate. In order to make a fair comparison between different arrays the failure rate must be escalated to reflect differences in complexity. By equating the reliability of rows in each strategy (equations (5.19) and (5.21)), and solving for the failure rate in the cell-elimination strategy (λ_C), it is possible to determine the failure rate for which cell-elimination yields the same MTBF as a row-elimination with failure rate λ_R. Figure 5.29 illustrates values of λ_C for varying numbers of redundant cells within an architecture.

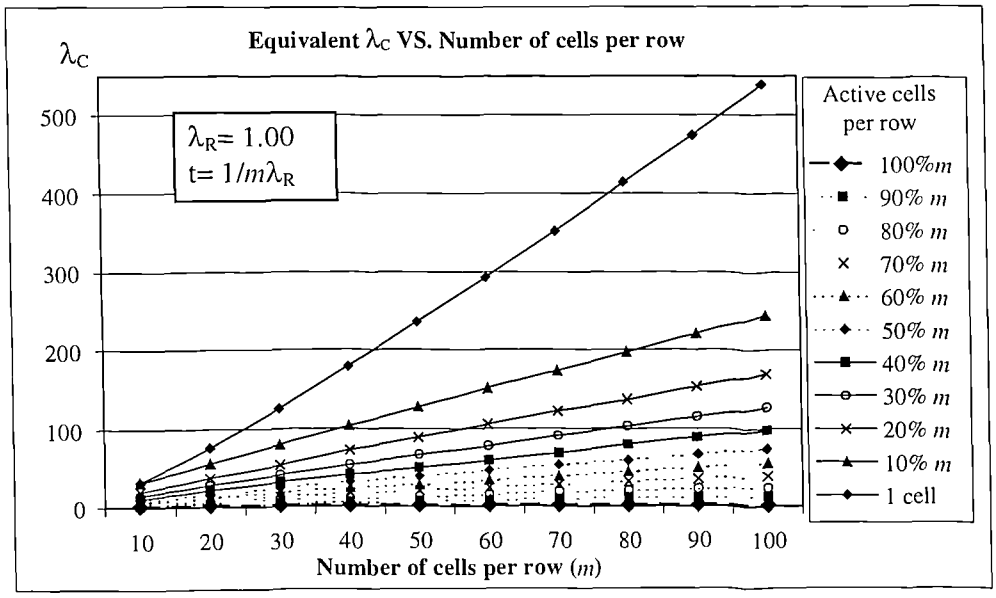


Figure 5.29 Equivalent failure rates for row- and cell-elimination

In figure 5.29, the graph for one active cell per row gives an upper bound to the complexity of cells in the cell-elimination strategy. Values of λ_C larger than these will produce cell-elimination architectures with lower overall reliabilities than the equivalent row-elimination architecture. Figure 5.29 allows a quantifiable measure to be made between two different design strategies. For example, a 100-column system using row-elimination will have better reliability than a 100-column system with 60 spares (40% of cells are active), if the failure rate λ_C is larger than $100\lambda_R$.

Another criterion to compare the failure rates of different architectures is to assume the failure rate proportional to the number of equivalent gates or transistors that constitute a cell. A detailed calculation of the failure rate is beyond the scope of the present work, but is an area that must be covered in future research.

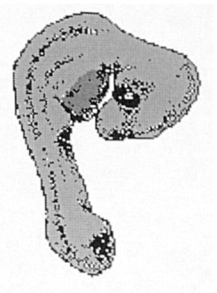
5.6 Summary

Embryonic arrays exploit hardware redundancy to achieve fault tolerance. Spare elements are incorporated at different levels of the embryonics hierarchy, achieving resilience of organisms to faults in their constituent cells. Mathematical models that represent reliability of embryonic arrays have been derived in this chapter. These models allow the reliability analysis of the embryonics architecture for different combinations of spare cells and molecules.

The reliability models presented have been derived by recursively applying the reliability models of series, parallel and k-out-of-m systems. The row- and cell-elimination reconfiguration strategies have been analysed following this methodology. It has been demonstrated that the reliability curves derived from the analysis can be used to compare the reliability of arrays with different parameters such as number of constituent cells, number of spare cells and different failure rates.

A special case of the embryonics project, called the MICTREE architecture, has also been analysed. By obtaining reliability models for the MICTREE hierarchy, it has been demonstrated that the methodology proposed can be extrapolated to analyse other cellular architectures with spares.

It has been verified that system reliability is improved by adding spare cells to the system. However, the rate of improvement in reliability is reduced as the number of spare cells increases. A point is reached where adding more spare cells to a system will not have a significant effect on system reliability. Therefore, a cost/benefit analysis based on the models proposed must be carried out to determine the optimum number of spare cells for a given application The models proposed can simplify this task.



CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 General Conclusions

This thesis has presented a novel paradigm for constructing fault-tolerant digital systems. The new paradigm, named embryonics, is the result of combining ideas and concepts borrowed from different fields. It has been shown that by providing cellular automata with biological attributes such as genome, reproduction, self-diagnosis and healing, the resulting system presents fault tolerance properties. The hypothesis that was stated in the introduction of this work has been confirmed: embryonic systems are viable alternatives for the design and implementation of fault-tolerant systems.

In chapter 1, the importance of fault tolerant computing systems was demonstrated by discussing to what extent modern societies depend on the use of computers for their stable functioning. A review of the development of fault tolerant computing since the early 40's allows us to conclude that the field has reached a point where new paradigms are needed in order to deal with the complexity and size of modern systems. Nature, through thousands of millions of years of evolution, has found a solution to the problem of fault tolerance in complex systems. Multicellularity, autonomous mechanisms to detect and combat infections, healing, inheritance of genetic information through DNA, embryonic development and learning are some of the mechanisms that guarantee, to certain extent, the survival of biological organisms. This thesis has demonstrated that by using Biology as an inspiration, it is possible to innovate in the field of fault-tolerant computing.

Bio-inspired systems and evolvable hardware were covered in chapter 2. Embryonics is a ramification of a broader field known as evolvable hardware (EHW). Although EHW deals mainly with circuits evolved, rather than designed, by means of genetic strategies, it is possible to regard embryonics as an evolvable architecture since an embryonic array

autonomously changes its configuration according to changes in the environment. The environment in this case is the internal state of the array, measured by the number of active cells. Autonomous reconfiguration by means of a genome is what distinguishes embryonic arrays from conventional array reconfiguration strategies.

Evolvable hardware is itself a branch of the research undertaken in Artificial Life (ALife). ALife pursues the creation of systems that recreate all, or at least some, of the features that characterise life. ALife investigates life not only as it is, but also as it could be.

Bio-inspired systems are classified according to the POE model. Embryonics belongs to the ontogenetic category, i.e. systems inspired by the development of multicellular systems in nature. Large hardware cellular systems are good platforms to investigate, in real time, the emergent behaviours characteristic of bio-inspired systems. Embryonics offers a good alternative for this kind of research.

In chapter 3 the technologies involved in the embryonics concept were outlined. The main ideas come from Biology. The embryonic development of multicellular organisms and the central dogma of molecular biology are at the core of the embryonics paradigm. Cellular automata provide the technical framework to transport multicellularity to the engineering domain and the latest generations of Field Programmable Gate Arrays (FPGAs), rapidly approaching the million-gate devices, provide the physical medium where embryonic arrays can be tested and applied. However, the ultimate goal is to integrate embryonic arrays in silicon and present them as a new family of bio-inspired FPGAs. Only then, all the benefits of the embryonics architecture will be truly exploited.

Chapter 4 presented a detailed description of the embryonic cell's architecture known as MUXTREE. The basic cell performs a selection function whose inputs and outputs are controlled by a configuration register. The regularity of the architecture makes it suitable to be implemented using state of the art FPGAs or WSI circuits. Furthermore, time to reconfiguration in MUXTREE arrays is determined by hardware, therefore embryonic arrays can be used in real-time applications where response time is a critical constraint. Embryonic arrays can also be used as a general-purpose tool to investigate the properties of cellular automata and array-based systems. Being a nascent discipline; much research must be undertaken to investigate in depth the real-time fault-tolerant properties of embryonic systems.

The penalty for gaining autonomous fast reconfiguration is, as in any fault-tolerant system, the amount of redundant resources required. Further work is needed to optimise the architecture of the basic cell and the reconfiguration process, in order to minimise both, the complexity of the cell and the amount of spare resources needed.

In this work two reconfiguration strategies were simulated and implemented: row-elimination and cell-elimination. These strategies are very efficient in terms of complexity and speed. However, the embryonics architecture can be extended to support more complex reconfiguration strategies. This effort would be justifiable only if the complexity of the processing unit was also increased; otherwise the complexity of the hardware supporting the reconfiguration mechanism would surpass that of the processing unit, becoming an overhead rather than a solution.

Three examples of the application of embryonic arrays were presented: A 3-input voter, an up-down 2-bit counter and a programmable frequency divider. These examples demonstrated that embryonic systems achieve fault tolerance by reconfiguring themselves using mechanisms found in biological embryonic systems. During the embryonic development of multicellular organisms, the failure (death) of one cell does not impact on the overall functionality of the tissue, organ or limb affected because of the tremendous amount of redundant elements available. If a cell or small group of cells die, healthy neighbours take over the missing function. Embryonic arrays are capable of successfully mimicking this mechanism.

Using embryonic arrays implies the mapping of a particular application onto the cells of the array, *i.e.* assigning a physical cell to every logical one. A methodology to systematically achieve this task was conceived during the development of this work. In this methodology the algorithm used to assign physical cells has to be different depending on the reconfiguration strategy implemented. For row-elimination, the algorithm must minimise the number of unused cells per row so that the number of spare rows can be maximised. For cell-elimination the maximum number of spare cells per row should be sought after.

The ultimate goal of incorporating fault tolerance into a system is to improve its reliability, i.e. the capability of the system to perform its expected job under the specified conditions of use over an intended period of time. To formally demonstrate fault tolerance in embryonic arrays, mathematical reliability models for the reconfiguration strategies presented were developed in chapter 5.

To set a theoretical framework, chapter 5 begins with an introduction to the main concepts in reliability analysis. For example, formal definitions for reliability and Mean Time Between Failures were given to support the mathematical analysis. The chapter continues with the development of reliability expressions for well-known system configurations, namely series, parallel, k-out-of-m and some combinations of them. Models for row-elimination and cell-elimination are derived by following a methodology that combines the mathematical models

of well-known structures. The graphs of the reliability models derived are useful to analyse and compare different configurations of arrays with spare components.

The methodology used to obtain reliability models for row- and cell- elimination was employed to analyse a third embryonics architecture called MICTREE. MICTREE is an embryonic hierarchical architecture with spare elements at different levels of the hierarchy. The reliability models presented in this work allow the analysis of the MICTREE architecture for different combinations of spare cells and molecules, the basic components of the MICTREE hierarchy.

It was verified that, in terms of reliability, the best way of colonising an array of molecules is to allocate active molecules column-wise. In this way, the number of spare columns can be maximised, with the corresponding improvement in reliability. At the organism level, reliability improves proportionally to the number of spare cells, but as the number of spare cells increases, the contribution to system reliability decreases. Therefore, a cost/benefit analysis must be carried out to determine the optimum number of spare cells for a given application.

The distributed automatic reconfigurability characteristic of embryonic arrays offers considerable advantages over more conventional reconfiguration strategies where, in most cases, a centralised agent, e.g. operating system or central processor, must solve the routing of information problem. For reliability analysis purposes, the effects that this central router imposes to the system must be taken into account. The central agent should be considered to be connected in series to the array, *i.e.* both must perform their functions correctly in order to consider the whole system in working order. Since the reliability model for a system whose components are connected in series involves the multiplication of the reliability expressions of each component, system reliability will always be lower than that of the element with the lowest reliability. This is true because the maximum value for reliability is 1, and from there it always decreases. Therefore, the centralised approach should be avoided for the design of highly dependable applications.

The reliability models presented in chapter 5 can be adapted to cellular systems other than embryonics. Further research must carried out in order to determine to what extent the models proposed hold for any fault-tolerant cellular system with spares.

The research will continue extending the present model towards the analysis of embryonic arrays with hundreds of thousands or even millions of cells. Large hardware cellular systems will be ideal platforms to investigate in real time the emergent behaviours characteristic of bio-inspired systems.

6.2 The Future of Embryonics

Embryonics is essentially an *experiment*, in the sense that the project was conceived for much more than to achieve a specific goal, but rather to look for insights by applying new concepts to a known field. One of its objectives was to investigate if interesting results can be obtained by applying concepts usually associated with biological processes to the design of computer hardware. However, the following points must be addressed if a practical application is to be implemented.

- There is a remarkable imbalance between the resources needed to implement the processing element (the selection function) and the resources needed to implement the reconfiguration mechanism (e.g. address calculation, memory and BIST). The reconfiguration mechanism occupies a large percentage of the silicon area required to implement the cell; hence the integration of several cells in one integrated circuit becomes impractical. Further research is needed to improve this balance.
- The BIST logic proposed in this work was selected giving preference to simplicity. However, if the complexity of the processing unit is increased, then the self-test mechanisms can be improved. For example, in the MICTREE architecture, where the processing unit is a microprogrammed Turing machine, self-test routines could be incorporated as part of the genome of each cell.
- The size, in silicon, of the memory needed in each cell is considerably bigger than the rest of the logic needed to implement an embryonic cell. To solve this imbalance it is necessary to optimise the storage of the genome on each cell. One possibility is to design a programmable look-up table (LUT) that receives the cell's co-ordinates as inputs and generates the corresponding configuration register (gene) in its outputs. This approach matches very well the internal architecture of some LUT-based commercial FPGAs.

Daniel Mange and his team at the Logic Systems Laboratory have followed an academic approach and have designed different versions of the original MUXTREE cell (their first embryonic implementation). They migrated from a completely static architecture based in a multiplexer, to a microprogrammed approach, the MICTREE architecture. The functionality remains that of a selector, however, instead of having different functional blocks to perform address calculation, reconfiguration and selection; they have designed a molecular-based Turing machine which, controlled by a microprogram, can execute all these functions. The result is that the new approach requires less silicon space, although the ratio configuration logic/functional logic (C/F) still remains too high. Their goal is to increase the complexity of

the functional block in order to simplify the reconfiguration and address calculation functions.

Pierre Marchal and his team at the Swiss Centre of Electronics and Microtechnique are also investigating the embryonics architecture. They have migrated towards a more complex architecture. In their proposal the functional block resembles a small microprocessor and the memory block consists only of the configuration register needed to program the corresponding cell. The global effect is that the ratio C/F is dramatically reduced, and therefore, physical implementation of the cell in silicon becomes feasible. The near future objective is to design a chip with several tens of cells, which can be used in applications involving parallel algorithms. They respond more to commercial and engineering demands rather than to a scientific interest.

Taking into account the work done by the Swiss, there are several alternatives for the continuation of this project. It is clear that the Configuration/Function ratio must be dramatically reduced in order to permit the physical implementation of the designs in a commercial FPGA. One alternative is to migrate towards microprogrammed architectures, i.e. design a data path and control unit that perform address calculation, logic functions and self-diagnosis. This approach seems to be very efficient in terms of C/F ratio, but has the disadvantage of requiring a complete redesign of the architecture for each application.

Following nature, it would be interesting to investigate different levels of cellular organisation in embryonic arrays. For example, solve a simple function using some basic cells and call that a cluster. Then, design a more complex function using clusters of cells as the basic building block and even design a higher level application relying on clusters of clusters of cells. The subsumption architecture model provides a framework to study this line of research [Kel94].

Biological multicellular systems work, with the exception of some specialised organs like the heart, asynchronously. Cells are autonomous and communication and synchronisation with other cells is carried out using chemical and electrical "hand shakings". Further research is needed to investigate asynchronous embryonic architectures. Asynchronous design is clock-free, therefore clock propagation problems are eliminated and power consumption is reduced. Throughput improves in most cases, because processing takes place at the maximum speed that the manufacturing technology allows.

All the embryonic arrays presented in this work use the von Neumann neighbourhood for calculating the following state of every cell. Using different neighbourhoods, e.g. Moore neighbourhood, is another interesting avenue to examine. However, considering more neighbours increases the complexity of the interconnection network and the reconfiguration

mechanisms. Optimisation of the C/F ratio is a prime objective of every embryonics proposal; therefore research must be done giving this goal a high priority.

Future work can apply the ideas presented in this work to larger problems and to incorporate other aspects of natural science into engineering design. The embryonics project is still in its infancy, however the results obtained from the "baby" are encouraging. The application of biological concepts to the design of fault-tolerant engineering systems will continue to evolve and grow, and more results are expected as the baby moves into adolescence.

6.3 The Future of Evolvable Hardware

Progress in evolvable hardware, in its present state, is constrained by the following facts:

- FPGAs remain as "the platform" to explore evolvable hardware. However, commercial FPGAs are not designed having evolvable hardware in mind. This fact was clearly manifest when Xilinx retired from the market the 6200 family of partial-reconfiguration devices, which were, at that moment, the alternative that every EHW researcher was choosing. New families of devices that allow partial reconfiguration on-line have appeared, like the Virtex® family from Xilinx or the 10K family from Altera, but the structure of the programming bitstreams has been kept secret, preventing the implementation of any realistic intrinsic evolution in hardware.
- The characteristics of CAD tools available today do not match the needs of the evolvable hardware community. Available tools like Foundation® from Xilinx, MAX+II® from Altera or WorkView® from ViewLogic, were created to assist the implementation of non-modifiable digital circuits using FPGAs. Modification of bitstreams by means of genetic algorithms is not supported.

The market dictates the development of new families of FPGAs. The only chance of having one of the big FPGA manufacturers mass-producing a family of FPGAs specially designed for EHW applications, is to develop an application that clearly expose evolvable hardware's mercantile potential. The search for such "killer" application has become the Holy Grail of the evolvable hardware community. A careful reading of the works presented in the EHW events that regularly take place around the world suggests that the search for the Holy Grail of evolvable hardware could be accomplished in the near future. The world of digital electronics is going to change when that happens.

6.4 The Future of Artificial Life

Any progress in Artificial Life requires the joint effort of researchers from areas as dissimilar as Mathematics, Biology, Psychology, Computer Science, and Electronics. In practice, however, this interaction is far from ideal. Research groups very rarely have experts in more than two areas. One reason for this lack of interaction could be the discrepancies in the language used within different fields. For example, the concept of evolution acquires different meanings when used by a biologist, a computer scientist and a mathematician. The differences in language reflect the differences in the goals that every specialist pursues. As the different disciplines advance toward their particular goals, there has to be a point where the differences between sciences begin to disappear. It is then, when true interdisciplinary collaboration will not be an option, but a necessity. Before that moment, ALife will remain an "interesting" field of study.

6.5 Final Thoughts

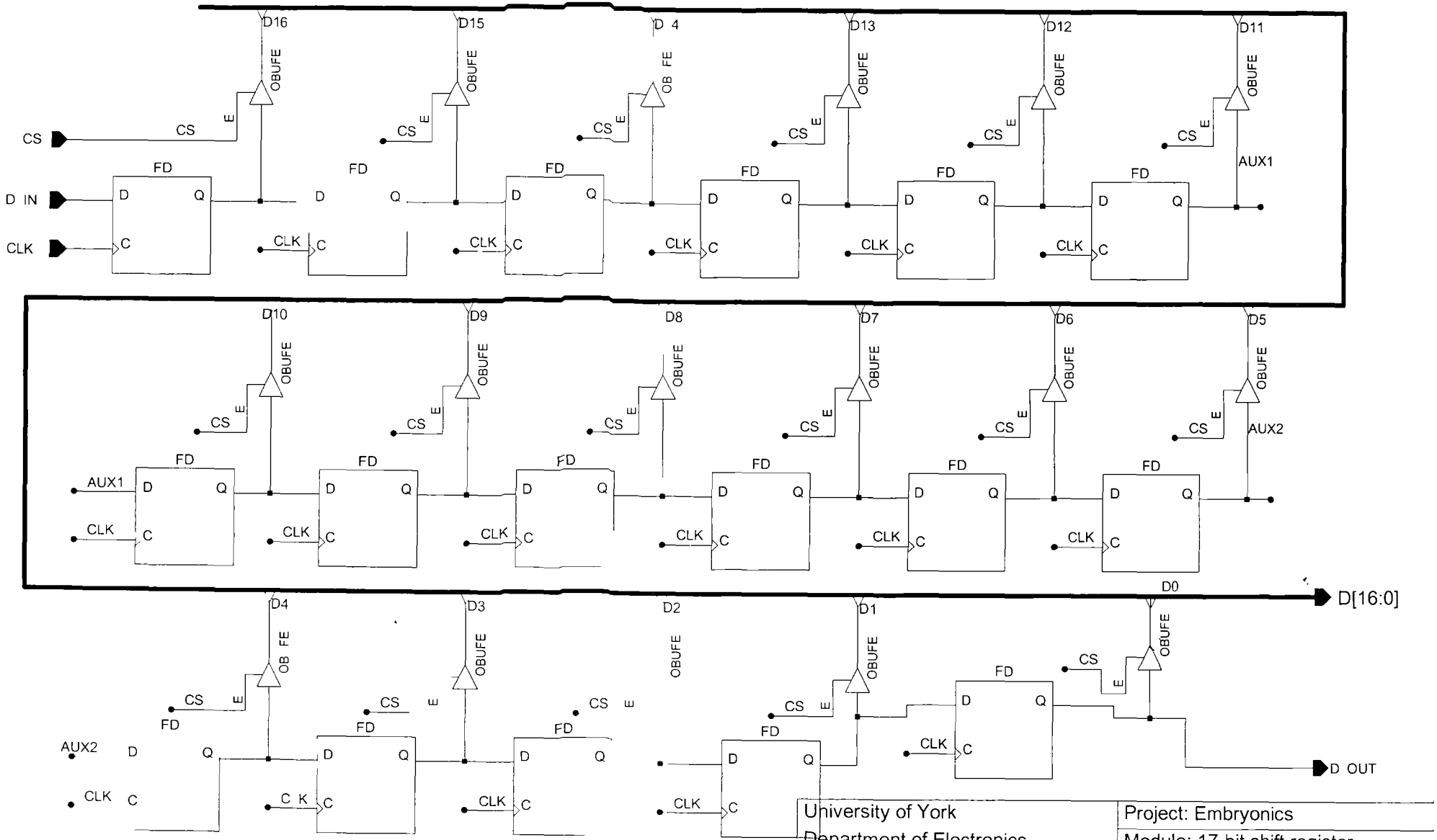
It is true that the idea of creating artificial life has accompanied men since the very beginning of history. However, it was not until the end of the 20th century that technology reached a point where artificial organisms could become something more than a blueprint. Advances in fields like Nanotechnology, Microelectronics, Micromechanics, Molecular Biology and Neuroscience are laying the foundations of the first truly artificial organism. Once achieved this objective, what the next step should be is not clear.

The first artificial being will not be intelligent, of course. Artificial intelligence is much harder to achieve than artificial life. Therefore, a possible next long-term goal could be the search for the artificial intellect. However, this quest implies moral and philosophical issues much deeper than those sparked off by artificial life. Will artificial individuals have rights? What will the ontological relationship between the creator and the created be? Are we destined to be slaves or gods of our own creations?

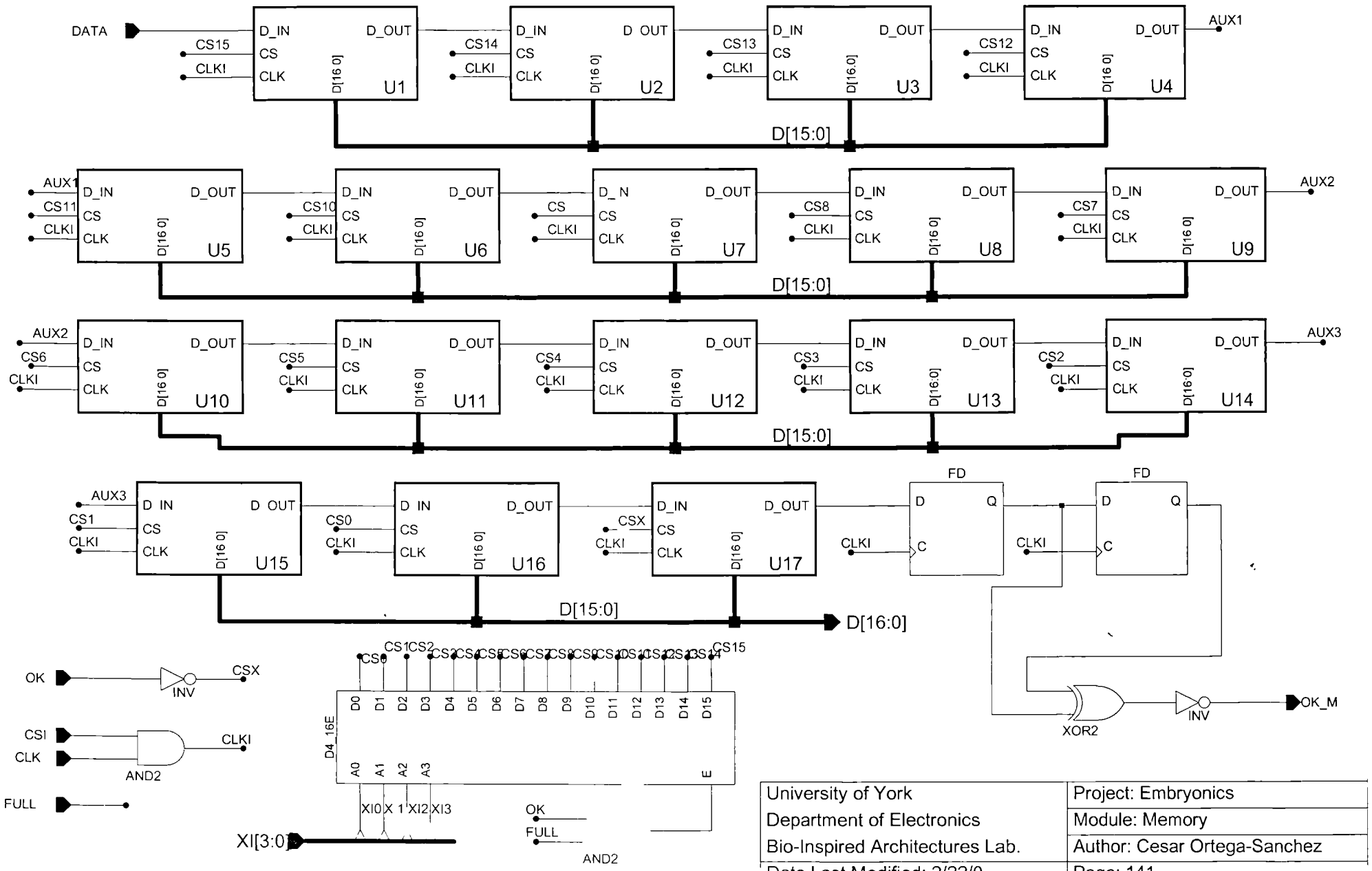
Whatever the physical features of the first artificial organism, it will represent the beginning of a new era where the natural and the artificial will co-exist. Are our societies up to the challenge of granting the category of "alive" to a system created by a group of men? Will fear and ignorance overpower science and technology? Will new generations of Nintendo kids accept the idea of an artificial being with the same indifference they receive what their parents call technological marvels? The answers to these questions will be found in the future which, in most of cases, is more amazing than our imagination.

APPENDIX

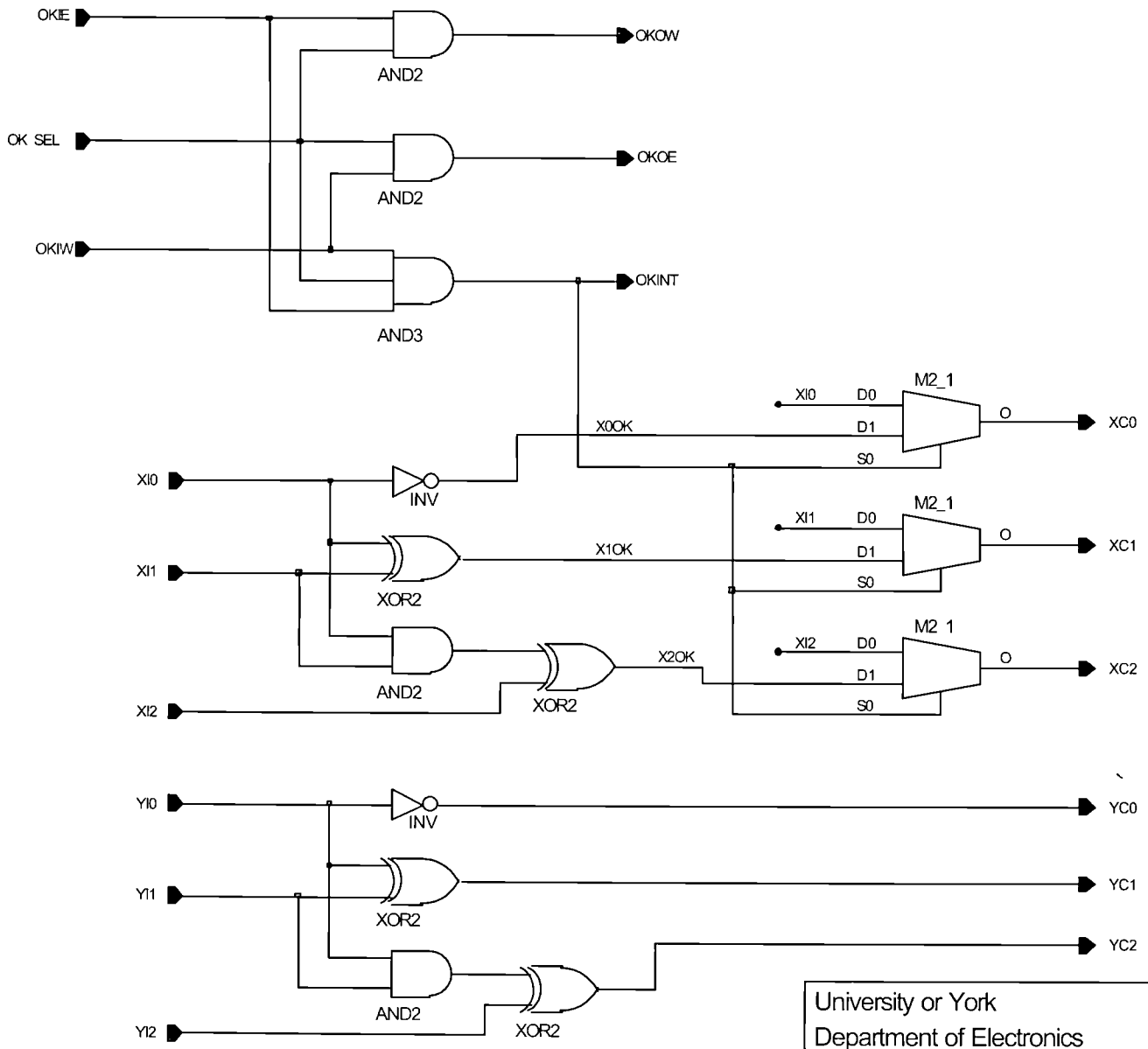
Schematic Diagrams of the Embryonic Cell



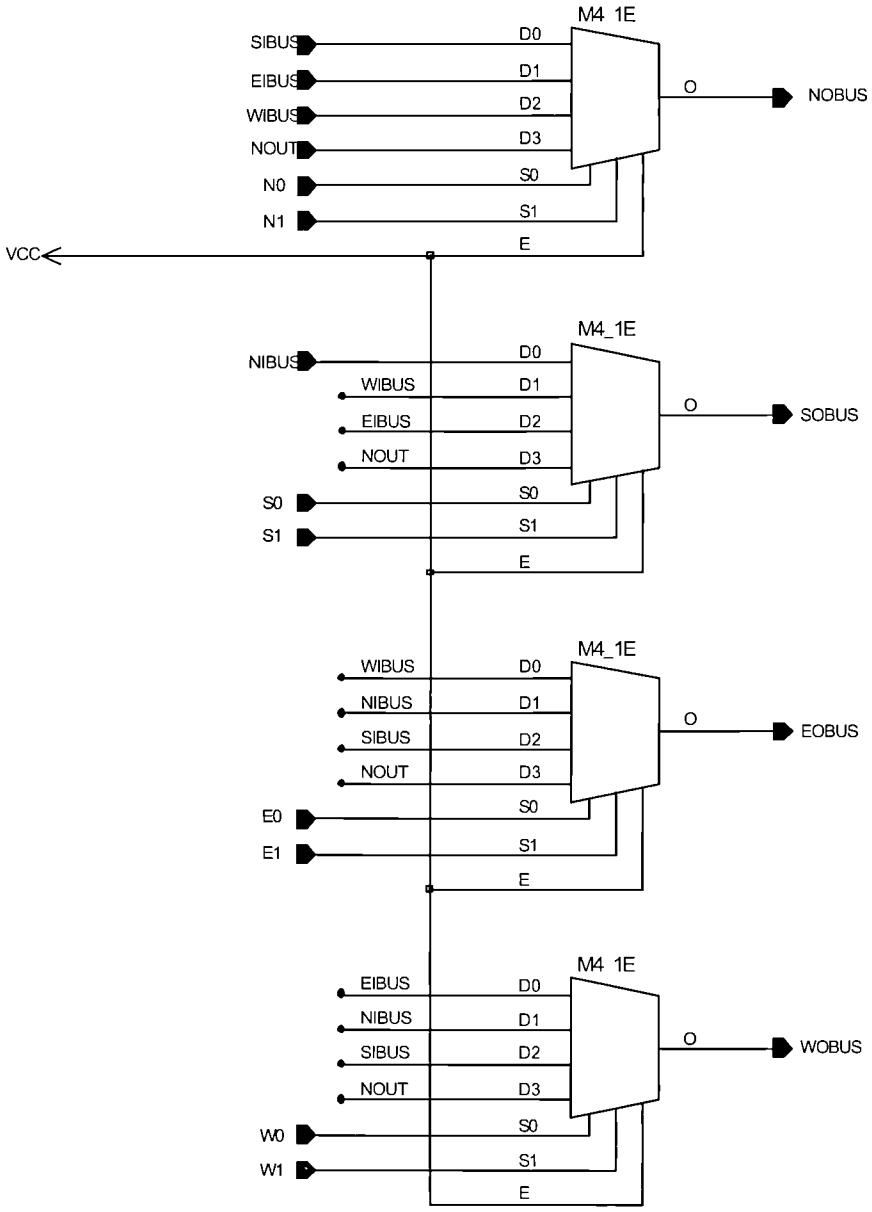
University of York	Project: Embryonics
Department of Electronics	Module: 17-bit shift register
Bio-Inspired Architectures Lab.	Author: Cesar Ortega-Sanchez
Date Last Modified: 2/22/0	Page: 140



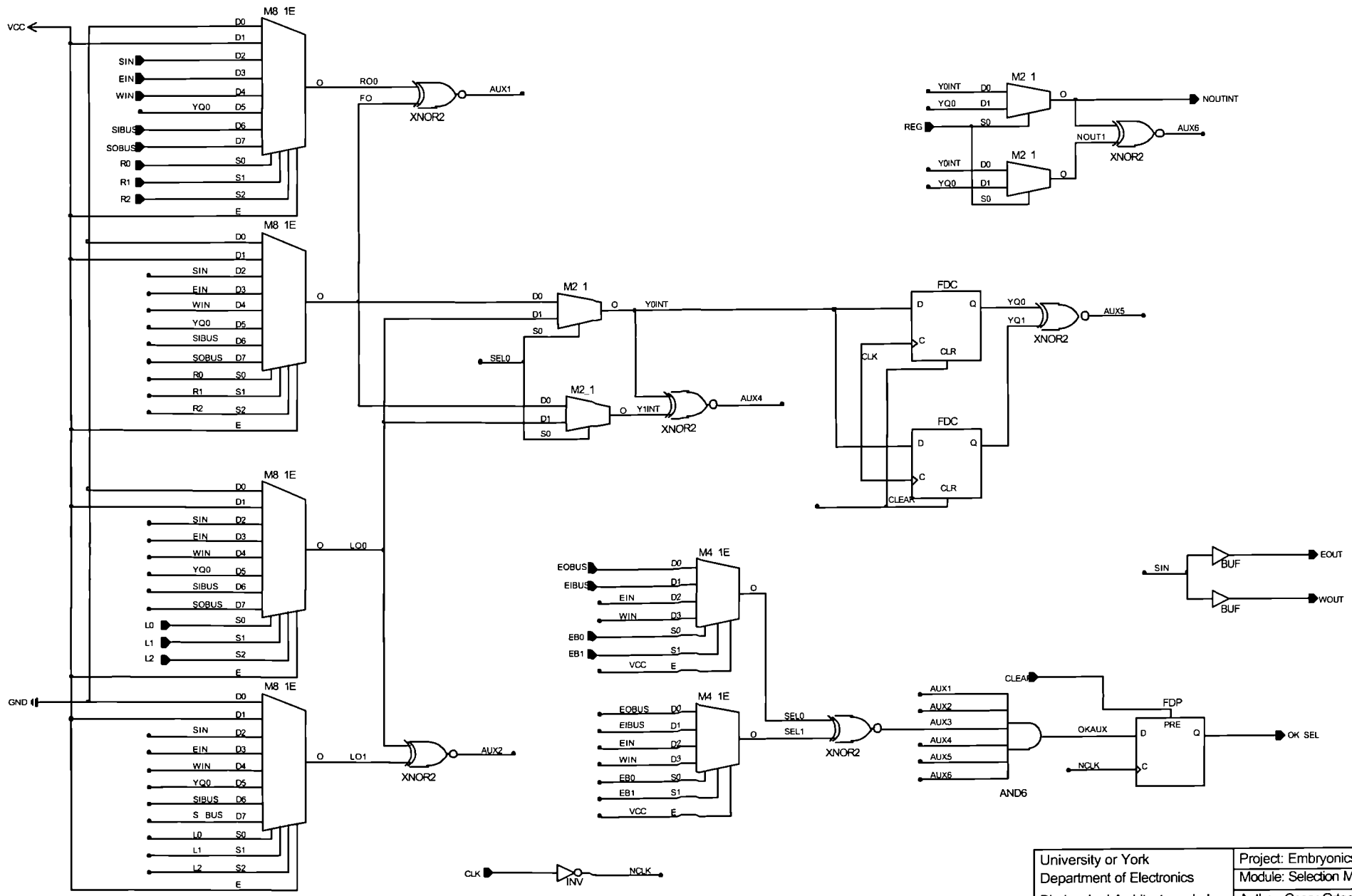
University of York	Project: Embryonics
Department of Electronics	Module: Memory
Bio-Inspired Architectures Lab.	Author: Cesar Ortega-Sanchez
Date Last Modified: 2/22/0	Page: 141



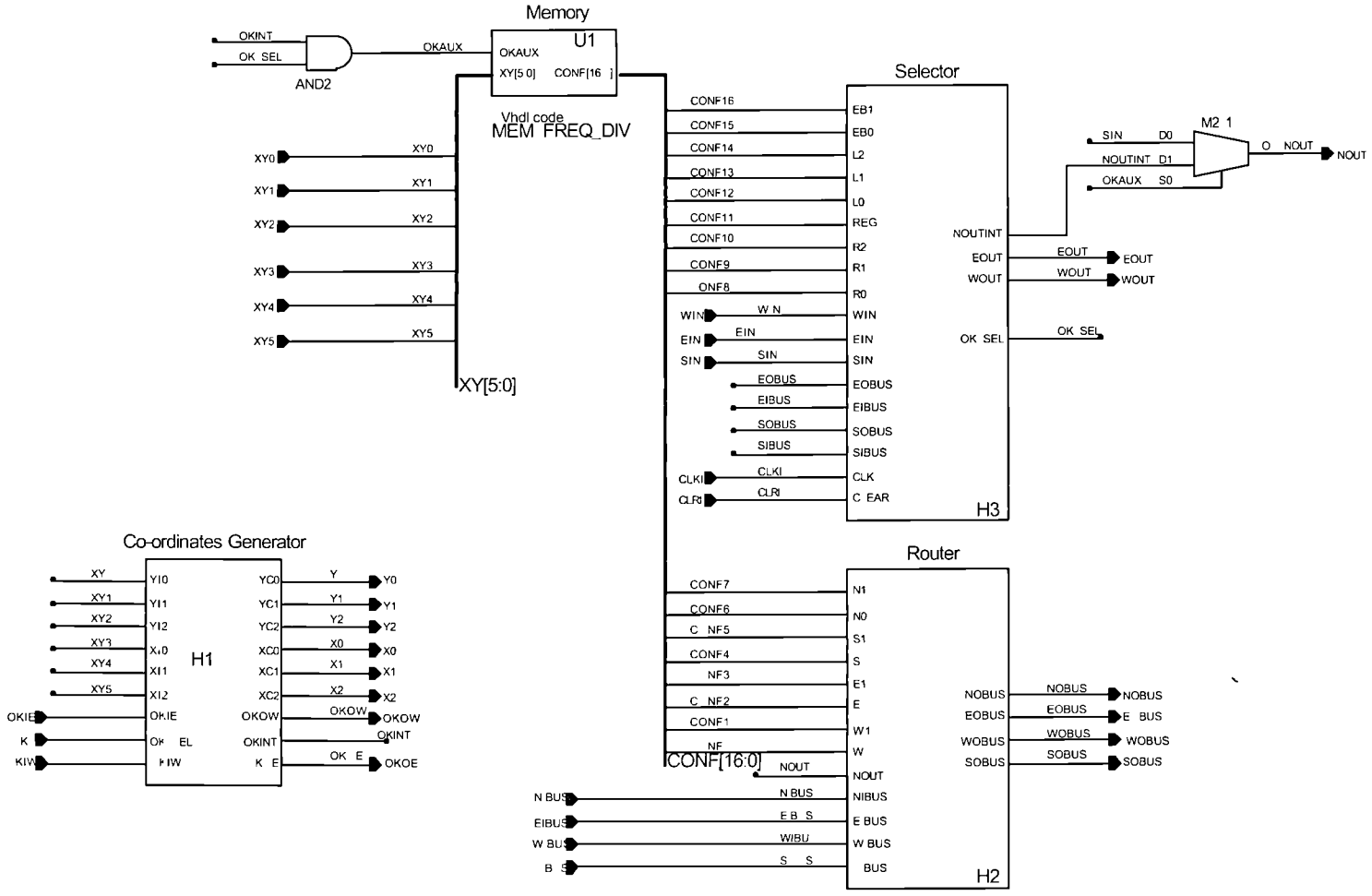
University or York	Project: Embryonics
Department of Electronics	Module: Co-ordinates Generator
Bio-Inspired Architectures Lab.	Author: Cesar Ortega-Sanchez
Date Last Modified: 2/22/0	Page: 142



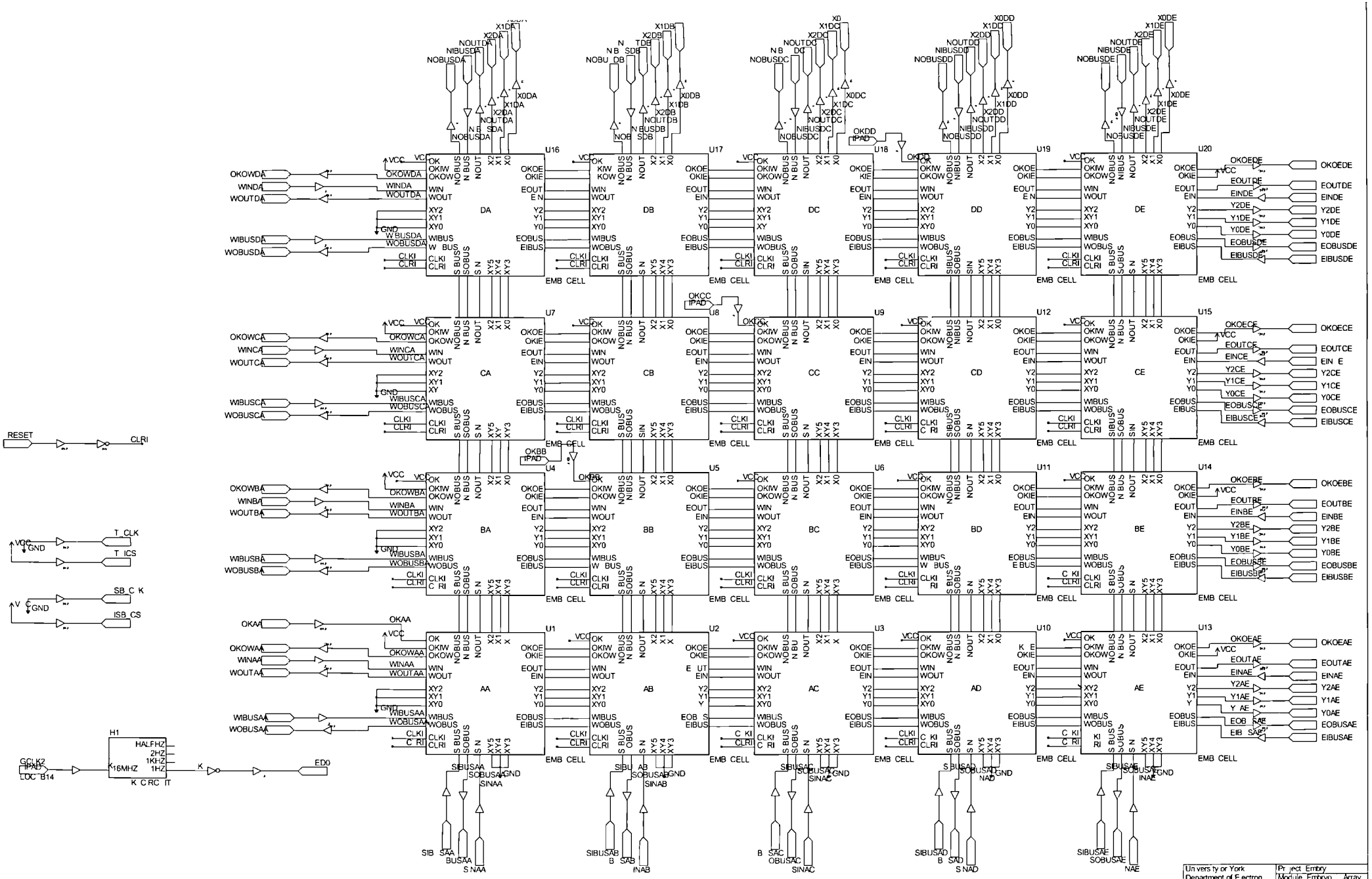
University or York	Project: Embryonics
Department of Electronics	Module: Router
Bio-Inspired Architectures Lab.	Author: Cesar Ortega-Sanchez
Date Last Modified: 2/22/0	Page: 143



University or York Department of Electronics Bio-Inspired Architectures Lab. Date Last Modified: 2/22/0	Project: Embryonics Module: Selection Module Author: Cesar Ortega-Sanchez Page: 144



University or York	Project: Embryonics
Department of Electronics	Module: Embryonic Cell
Bio-Inspired Architectures Lab.	Author: Cesar Ortega-Sanchez
Date Last Modified: 2/22/0	Page: 145



Un vers ty or York	Pr ject Embry
Department of E ectron	Module Embryo Array
B o-Insp red Arch tectures	Author Cesar Ortega-Sanc
Da'e Las Mod'ed 5/16/0	Page 146

BIBLIOGRAPHY AND REFERENCES

[Abr87] Abraham J., Banerjee P., Chen C. *et.al.*,"Fault Tolerance Techniques for Systolic Arrays", IEEE Computer, Vol.20-7, July 1987, pp.65-74

[Act95] Actel, FPGA Databook and Design Guide, Actel, Sunnyvale, 1995

[Ae97] Ae T., Fukumoto H. and Hiwatashi S., "Special-Purpose Brainware Architecture for Data Processing", in Higuchi T., Iwata M. and Liu W. (Eds.), Proceedings of 1st International Conference on Evolvable Systems: From Biology to Hardware (ICES96), LNCS 1259, Springer-Verlag, 1997, pp.289-301

[Ake78] Akers S., "Binary Decision Diagrams", IEEE Trans. on Comp., Vol.27-6, June 1978, pp.509-516

[All90] Allworth S. and Zobel R., Introduction to Real-time Software Design, Macmillan, Hong Kong, 1990

[And72] Anderson P., "More is Different", Science, Vol.177-4047, August, 1972, pp.393-396

[And88] Anderson P., Arrow K. and Pines D., The Economy as an Evolving Complex System, Addison-Wesley, 1988

[Avi78] Avizienis A., "Fault-Tolerance.- The survival attribute of digital systems", Procs. IEEE, Vol.66, Num.10, Oct., 1978

[Avi97] Avizienis A., "Toward Systematic Design of Fault-Tolerant Systems", IEEE Computer, Vol.30-4, April, 1997, Computer Society Press, pp. 51-58

[Ban94] Banatre M. and Lee P. (eds.), Hardware and Software Architectures for Fault Tolerance, LNCS 774, Springer-Verlag, 1994

[Bar89] Barbour A. and Wojcik A., "A General Constructive Approach to Fault-Tolerant Design Using Redundancy", IEEE Trans. on Computers, Vol.38-1, Jan. 1989, pp.15-29

[Bar92] Baron R. and Higbie L., Computer Architecture, Addison-Wesley, 1992

[Bas95] Bass J., Voting in Real-Time Distributed Computer Control Systems, PhD Thesis, University of Sheffield, Dept. of Automatic Control and Systems Engineering, Sept. 1995

[Bel78] Bell D., "Decision Trees, Tables and Lattices", in Batchelor B. (ed.), Pattern recognition: ideas in practice, Plenum Press, 1978.

[Bel88] Bellcore, TR-TSY-000332, Reliability Prediction Procedure for Electronic Equipment, Issue 2, July 1988.

[Bel92] Belkhale K. and Banerjee P., "Reconfiguration Strategies for VLSI Processor Arrays and Trees Using a Modified Diogenes Approach", IEEE Trans. on Computers, Vol.41-1, January 1992, pp.83-96

[Ben97] Bennett F., Koza J., Andre D. and Keane M., "Evolution of a 60dB Op Amp Using Genetic Programming", in Higuchi, Iwata and Liu (eds.), Proceedings of 1st International Conference on Evolvable Systems: From Biology to Hardware (ICES96), LNCS 1259, Springer-Verlag, 1997, pp.455-469

[Bet98] Betz V. and Rose J., "How Much Logic Should Go in an FPGA Logic Block?", IEEE Design and Test of Computers, January-March 1998, pp.10-15

[Bod96] Boden M.(ed.), The Philosophy of Artificial Life, Oxford Readings in Philosophy, Oxford University Press, 1996

[Bow92] Bowles J., "A Survey of Reliability-Prediction Procedures For Microelectronic Devices", IEEE Trans. on Reliability, Vol.41-1, March, 1992, pp.2-12

[Bre76] Breuer M., Diagnosis and Reliable Design of Digital Systems, Computer Science Press, 1976

[Bru97] Tutorial on Cellular Automata, Brunel University, http://www.brunel.ac.uk:8080/depts/AI/alife

[Bry86] Bryant R., "Graph-based Algorithms for Boolean Function Manipulation", IEEE Trans. on Computers, Vol.35-8, Aug.1986, pp.677-691

[Bry92] Bryant R., "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams", ACM Computing Surveys, Vol.24-3, Sep.1992, pp.293-318

[Bth84] British Telecom, Handbook of Reliability Data for Components Used in Telecommunications Systems, Issue 4, January 1984.

[Bur70] Burks A., Essays on Cellular Automata, 1970

[Bur84] Burks C., "Towards Modeling DNA Sequences as Automata", Physica 10D, 1984, pp.157-167

[Byl89] Byl J., "Self-Reproduction in Small Cellular Automata", Physica D, Vol. 34, 1989, pp.295-299

[Car84] Carter F., "The Molecular Device Computer: Point of departure for large scale cellular automata", Physica 10D, 1984, pp.175-194

[Cas97] McCaskill J., Maeke T., Gemm U., Schulte L. and Tangen U., "NGEN: A Massively Parallel Reconfigurable Computer for Biological Simulation: Towards a Sself-Organizing Computer", in Higuchi, Iwata and Liu (eds.), Evolvable Systems: From Biology to Hardware, LNCS 1259, Springer-Verlag, 1997, pp.260-276

[Cat99] Ctell K., Zhang S. and Serra M., "2-by-n Hybrid Cellular Automata with Regular Configuration: Theory and Application", IEEE Trans. on Computers, Vol.48-3, March 1999, pp.285-295

[Cer79] CernyE., "Synthesis of Minimal Binary Decision Trees", IEEE Trans. on Computers, Vol.28-7, July 1979, pp.472-482

[Cha70] Chang H., Fault Diagnosis of Digital Systems, Wiley, 1970,

[Che90a] Chean M. and Fortes J., "The Full Use of Suitable Spares (FUSS) Approach to Hardware Reconfiguration for Fault-Tolerant Processor Arrays, IEEE Trans. on Computers, Vol.39-4, April 1990, pp.564-571

[Che90b] Chean M. and Fortes J., "A Taxonomy of Reconfiguration Techniques for Fault-Tolerant Processor Arrays", IEEE Computer, Vol.23-1, January, 1990, pp. 55-69

[Che97] Chen Y., Upadhyaya S., Cheng C., "A Comprehensive Reconfiguration Scheme for Fault-Tolerant VLSI/WSI Array Processors", IEEE Trans. on Computers, Vol.46-12, December 1997, pp.1363-1371

[Chi94] Chillarege R., "Top Five Challenges Facing the Practice of Fault Tolerance", in [Ban94]

[Cho97] Chou T., "Beyond Fault Tolerance", IEEE Computer, Vol.30-4, April 1997, pp.47-49

[Chr80] Christensen B., Krausz H. and Perez-Polo R., "Communication Among Neurons and Neuroscientists", in Pinsker H. and Willis W. (Eds.), Information Processing in the Nervous System, Raven Press, 1980, pp.339-359

[CNE83] Centre National d'Etudes des Telecommunications (CNET), Recueil de Donnes de Fiabilite du CNET (Collection of Reliability Data from CNET), 1983

[Cod68] Codd E., Cellular Automata, Academic Press, 1968

[Dar72] Darwin C., The Origin of Species, Collier Brooks, 1872

[Dav94] Davis N., Gray G. and Wegner J., "Reconfiguring Fault-Tolerant Two-Dimensional Array Architectures", IEEE Micro, April 1994, pp.60-68

[Daw87] Dawkins R., "The Evolution of Evolvability", in Artificial Life, Langton C. (Ed.), Addison-Wesley, 1987

[Deh95] DeHon A., Tau E., Chen D., Eslick I. and Brown J., "A first generation of DPGA implementation", Procs. of the third Canadian workshop on Field-programmable devices, May 1995, pp.138-143

[Den98] Denson W., "The History of Reliability Prediction", IEEE Tr. on Reliability, Vol.47-3-SP, September, 1998, pp.SP321-328

[Dre90] Drexler E., Engines of Creation: The coming era of nanotechnology, Fourth State-London, 1990

[Dut92] Dutt S. and Hayes J., "Some Practical Issues in the Design of Fault-Tolerant Multiprocessors", IEEE Trans. on Computers, Vol.41-5, May 1992, pp.588-598

[Dut97] Dutt S. and Mahapatra N., "Node-covering, Error-correcting Codes and Multiprocessors with Very High Average Fault Tolerance", IEEE Trans. on Computers, Vol.46-9, 1997, pp.997-1014

[Dye95] Dyer M., "Toward Synthesising Artificial Neural Networks that Exhibit Cooperative Intelligent Behaviour: Some open issues in Artificial Life", in Langton C. (ed.), Artificial Life: an Overview, MIT Press, 1995, pp.111-134

[Eic65] Eichelberger E., "Hazard Detection in Combinational and Sequential Switching Circuits", IBM Journal, March 1965, pp.90-99

[Eva98] Evans R., "Electronics Reliability: A Personal View", IEEE Tr. on Reliability, Vol.47-3-SP, September, 1998, pp.SP-329-332

[Far84] Farmer D., Toffoli T. and Wolfram S. (eds.), Cellular Automata, North-Holland Physics, 1984

[Fer89] Ferris-Prabhu A., "Defects, Faults and Semiconductor Device Yield", in Koren I.(Ed.), Defect and Fault Tolerance in VLSI Systems, Plenum Press, 1989, Ch.3, pp.33-45

[Fis85] Fisher A. and Kung H., "Sinchronizing Large VLSI Processor Arrays", IEEE Trans. on Computers, Vol.34-8, August 1985, pp.734-740

[For85] Fortes J. and Raghavendra C, "Gracefully Degradable Processor Arrays", IEEE Trans. on Computers, Vol.34-11, 1985, pp.1033-1043

[For87] Fortes J. and Wah B., "Systolic Arrays- From Concept to Implementation", IEEE Computer, Vol.20-7, July 1987, pp.12-17

[Gar93] Garis de H., "Evolvable Hardware: Genetic Programming of a Darwin Machine", in Albrecht R., Reeves C. and Steele N. (Eds.), Artificial Neural Nets and Genetic Algorithms, Springer-Verlag, 1993, pp.441-449

[Gar96] Garis de H., "CAM-BRAIN: The evolutionary engineering of a billion neuron artificial brain by 2001", in Sanchez E. and Tomassini M. (Eds.), Towards Evolvable Hardware: The evolutionary engineering approach, LNCS 1062, Springer-Verlag, 1996, pp.76-98

[Gei90] Geist R. and Trivedi K., "Reliability Estimation of Fault-Tolerant Systems: Tools and Techniques", IEEE Computer, Vol.23-7, July 1990, pp.52-61

[Ger89] Gerhardt M., "A Cellular Automaton Describing the Formation of Spatially Ordered Structures in Chemical Systems", Physica D 36, 1989

[Ger97] Gers F. and de Garis H., "CAM-BRAIN: A new model for ATR's cellular automata based artificial brain project", in Higuchi, Iwata and Liu (eds.), Evolvable Systems: From Biology to Hardware, LNCS 1259, Springer-Verlag, 1997, pp.437-452

[Gle94] Gleeson B., "Fault-Tolerance: Why should I pay for it?", in [Ban94]

[Goe97] Goeke M., Sipper M., Mange D., Stauffer A., Sanchez E. and Toassini M., "Online Autonomous Evolware", in Higuchi, Iwata and Liu (eds.), Evolvable Systems: From Biology to Hardware, LNCS 1259, Springer-Verlag, 1997, pp.96-106

[Gok90] Gokhale M. and Minnich R., "Splash: A reconfigurable linear logic array", Intl. Conference on Parallel processing, 1990, pp.526-532

[Gos93] Goseva K., "N-version programming with Majority Voting Decision: Dependability modelling and evaluation", on Microprocessing and Microprogramming, Vol.38, pp.811-818, 1993

[Grh97] Gerhart J. and Kirschner M., Cells, Embryos and Evolution, Blackwell Science, 1997

[Gro94] Grosspietsch K., "Fault Tolerance in Highly Parallel Hardware Systems", IEEE Micro, Feb. 1994, pp.60-68

[Hae97] Haenni J., "Hardware Implementation of von Neumann's Automaton", presented at the von Neumann's Day organised by the LSL at the EPFL, 25th July, 1997

[Han77] Hanani M., "An Optimal Evaluation of Boolean Expressions in an online query system", Commun. ACM, Vol.20-5, May 1977, pp.344-347

[Har96] Harvey I., Husbands P., Cliff D., Thompson A. and Jakobi N., "Evolutionary Robotics at Sussex", School of Cognitive and Computing Sciences, University of Sussex, 1996, http://www.cogs.susx.ac.uk/users/inmanh

[Har97] Harvey I. and Thompson A., "Through the Labyrinth Evolutions Finds a Way: A silicon ridge", in Higuchi, Iwata and Liu (eds.), Evolvable Systems: From Biology to Hardware, LNCS 1259, Springer-Verlag, 1997, pp.406-422

[Hau97] Hauck S., "The roles of FPGAs in Reprogrammable Systems", Submitted to Proceedings IEEE, 1997

[Hay76] Hayes J., "A Graph Model for Fault-Tolerant Computing Systems", IEEE Trans. on Computers, Vol.25-9, September 1976, pp.875-884

[Hem96] Hemmi H., Mizoguchi J. and Shimohara K., "Development and Evolution of Hardware Behaviours", in Sanchez E. and Tomassini M. (Eds.), Towards Evolvable Hardware: The evolutionary engineering approach, LNCS 1062, Springer-Verlag, 1996, pp.250-265

[Hig96] Higuchi T., Iwata M., Kajitani I. and Iba H., "Evolvable Hardware and its Application to Pattern Recognition and Fault-Tolerant Systems", in Sanchez E. and Tomassini M. (Eds.), Towards Evolvable Hardware: The evolutionary engineering approach, LNCS 1062, Springer-Verlag, 1996, pp.118-135

[Hig97] Higuchi T, Iwata M. and Liu W. (eds.), Evolvable Systems: From Biology to Hardware, LNCS 1259, Springer-Verlag, 1997

[Hig99] Higuchi T. and Kajihara N., "Evolvable Hardware Chips for Industrial Applications", Communications of the ACM, Vol.42-4, April 1999, pp.60-66

[Hik97] Hikage T., Hemmi H. and Shimohara K., "Hardware Evolution System: Introducing dominant and recessive heredity", in Higuchi, Iwata and Liu (eds.), Evolvable Systems: From Biology to Hardware, LNCS 1259, Springer-Verlag, 1997, pp.423-437

[Hil84] Hillis D., "The Connection Machine: A computer architecture based on cellular automata", Physica 10D, 1984, pp.213-228

[Hin87] Hinton G. and Nowlan S., "How Learning Can Guide Evolution", Complex Systems, Vol. 1, 1987

[Hog92] Hogeweg, P., "As Large as Life and Twice as Natural: Bioinformatics and the Artificial Life Paradigm", in D G. Green and T. J. Bossomaier (eds.) Complex systems: From Biology to Computation, IOS Press, pp.2-10

[Hol82] Hollaar L., "Direct Implementation of Asynchronous Control Units", IEEE Trans. on Computers, Vol.31-12, December 1982, pp.1133-1141

[Hol92] Holland J., Adaptation in Natural and Artificial Systems, MIT Press, Cambridge, MA, 1992

[Hut95] Hutchings B., "A Dynamic Instruction Set Computer", in Procs. of the IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society, April 1995

[Hwa85] Hwang K. and Briggs F., Computer Architecture and Parallel Processing, McGraw-Hill, 1985

[IEC74] IEC Document: List of Basic Terms, Definitions and Related Mathematics for Reliability, Publication No.271, International Electrotechnical Commission, Geneva, Switzerland, 1974

[Ino98] Inoue T., Miyazaki S. and Fujiwara H., "Universal Fault Diagnosis for Lookup Table FPGAs", IEEE Design and Test of Computers, January-March 1998, pp.39-44

[Ish97] Ishida Y., "The Immune System as a Prototype of Autonomous Decentralized Systems: An Overview", IEEE, pp.85-92

[Jay93] Jay C., "VHDL and Synthesis Tools provide a Generic Design Entry Plarform into FPGAs, PLDs and ASICs", Microprocessors and Microsystems, Vol.17-7, Sept. 1993

[Joh89] Johnson B., Design and Analysis of Fault-Tolerant Digital Systems, Addison-Wesley, 1989

[Kau96] Kauffman S., At Home in the Universe: The Search for Laws of Self-Organization and Complexity, Penguin Books, 1996

[Kel94] Kelly K., Out of Control: The new Biology of machines, Fourth State-London, 1994

[Kep94] Kephart J., "A Biologically Inspired Immne System for Computers", in Brooks R. (ed.), Artificial Life IV, MIT Press, 1994, pp.130-139.

[Key97] Keymeulen D., Durantez M and Hoshino T., "An Evolutionary Robot Navigation System Using a Gate-Level Evolvable Hardware", in Higuchi, Iwata and Liu (eds.), Proceedings of 1st International Conference on Evolvable Systems: From Biology to Hardware (ICES96), LNCS 1259, Springer-Verlag, 1997, pp.195-209

[Kir96] Kirk R., "CAD Using ViewLogic and Lattice ISP PLDs", Course Notes, Deparment of Electronics, University of York, 1996

[Kit96] Kitano H., "Morphogenesis for Evolvable Systems", in Sanchez E. and Tomassini M. (Eds.), Towards Evolvable Hardware: The evolutionary engineering approach, LNCS 1062, Springer-Verlag, 1996, pp.99-117

[Kit97] Kitano H., "Challenges of Evolvable Systems: Analysis and future directions", in Higuchi, Iwata and Liu (eds.), Evolvable Systems: From Biology to Hardware, LNCS 1259, Springer-Verlag, 1997, pp.125-135

[Kor89] Koren I.(Ed.), Defect and Fault Tolerance in VLSI Systems, Plenum Press, 1989

[Kor90] Koren I. and Singh D., "Fault Tolerance in VLSI Circuits", IEEE Computer, Vol.23-7, IEEE Computer Society, July 1990, pp.73-83

[Koz96] Koza J., Goldberg D., Fogel D. and Riolo R. (eds.), Genetic Programming 1996: Proceedings of the First Annual Conference, MIT Press, 1996.

[Ku97] Ku H. and Hayes J., "Systematic Design of Fault-Tolerant Multiprocessors with Shared Buses", IEEE Trans. on Computers, Vol.46-4, April 1997, pp.439-455

[Kng82] Kung H., "Why Systolic Architectures?", IEEE Computer, Vol.15-1, Jan. 1982, pp.37-46

[Kun82] Kung S., Arun K. and Gal-Ezer R., "Wavefront Array Processor: Language, Architecture and Applications", IEEE Trans. on Computers, Vol.31-11, November 1982, pp.1054-1066

[Kun87] Kung S., Lo S. and Jean S., "Wavefront Array Processors- Concept to Implementation", IEEE Computer, Vol.20-7, July 1987, pp.18-33

[Kun89] Kung S., Jean S. and Chang C., "Fault-Tolerant Array Processors Using Single-Track Switches", Trans. on Computers, **Vol.38-4**, 1989, pp 501-513

[Kuo87] Kuo S. and Fuchs K., "Efficient Spare Allocation for Reconfigurable Arrays", IEEE Design and Test of Computers, Vol.4, February 1987, pp.24-31

[Lal85] Lala P., Fault Tolerance and Fault Testable Hardware Design, Prentice-Hall, 1985

[Lal96] Lala P., Practical Digital Logic Design and Testing, Prentice-Hall, 1996

[Lal97] Lala P., Digital Circuit Testing and Testability, Academic Press, 1997

[Lan84] Langton C., "Self-reproduction in Cellular Automata", Physica 10D, 1984, pp.135-144

[Lan89] Langton C., "Artificial Life", 1989, Reprinted in Boden M. (ed.), The Philosophy of Artificial Life, Oxford University Press, 1996, pp.39-94.

[Lan95] Langton C. (ed.), Artificial Life: an Overview, MIT Press, 1995

[Lap90] Laprie J.C., Arlat J., Béounes C. and Kanoun K., "Definition and Analysis of Hardware and Software Fault-Tolerant Architectures", Computer Vol.23-7, IEEE Computer Society, July 1990, pp.39-51

[Lap92] Lapric J.C. (ed.), Dependability: Basic concepts and terminology, IFIP WG 10.4, Springer-Verlag, Wien-NewYork, 1992

[Law82] Lawrence S., "Introduction to the Configurable Highly Parallel Computer", IEEE Computer, Vol.15-1, January 1982, pp.47-55

[Lee59] Lee C., "Representation of Switching Circuits by Binary-Decision Programs", Bell Systems Technical Journal, July 1959, pp.985-999

[Lee90] Lee P. and Anderson T., Fault-Tolerance: Principles and practice, Springer-Verlag, Wien-Ney York, 1990

[Lei85] Leighton T. and Leiserson C., "Wafer-Scale Integration of Systolic Arrays", IEEE Trans. on Computers, Vol.34-5, May 1985, pp.448-461

[Lia92] Liaw H. and Lin C., "On the OBDD-Representation of General Boolean Functions", IEEE Trans. on Computers, Vol.41-6, June 1992, pp.661-664

[Lid94] Liddell D., "Simple Design Makes Reliable Computers", in [Ban94]

[Lim88] Lim H., "Lattice Gas Automata of Fluid Dynamics for Unsteady Flow", Complex Systems, Vol. 2, pp.45-68, 1988

[Loh99] Lohn J., "Experiments on Evolving Software Models of Analog Circuits", Communications of the ACM, Vol.42-4, April 1999, pp.67-69

[Lom89] Lombardi F., Sami M. and Stefanelli R., "Reconfiguration of VLSI Arrays by Covering", IEEE Trans. on CAD, Vol.8-9, September 1989, pp.952-965

[Lon80] Longbottom R., Computer System Reliability, John Wiley & Sons, 1980

[Lut90] Luthra Puran, "Mil-Hdbk-217: What is Wrong with it?", IEEE Tr. on Reliability, Vol.39-5, December, 1990, p.518

[Mad89] Madre J., Coudert O. and Billon J., "Automating the Diagnosis and Rectification of Design Errors with PRIAM", in Procs. of Intl. Conf. on CAD, IEEE, Santa Clara, CA, November 1989, pp.30-33

[Man92] Mange D., Microprogrammed Systems, Chapman and Hall, 1992

[Man95a] Mange D., Durand S., Sanchez E., Stauffer A., Tempesti G., Marchal P. and Piguet C., "A new Paradigm for Developing Digital Systems Based on a Multi-cellular Organisation", Technical report 95/115, EPFL, Logic Systems Laboratory, April 1995

[Man95b] Mange D., Sanchez E., Stauffer A., Tempesti G., Durand S., Marchal P. and Piguet C., "Embryonics: A new methodology for designing FPGAs with self-repair and self-

reproducing properties", Technical report 95/152, EPFL, Logic Systems Laboratory, 1995

[Man96a] Mange D., Goeke M., Madon D., Stauffer A., Tempesti G. and Durand S., "Embryonics: A new family of coarse-grained FPGA with self-repair and self-reproduction properties", in Sanchez E. and Tomassini M. (Eds.), Towards Evolvable Hardware: The evolutionary engineering approach, LNCS 1062, Springer-Verlag, 1996, pp.197-220

[Man96b] Mange D., Madon D., Sanchez E., Stauffer A., Tempesti G., Durand S., Marchal P. and Piguet C., "BIOWATCH: Une montre autoréparable et autoreproductrice", Technical report 96/186, EPFL, Logic Systems Laboratory, May 1996

[Man97a] Mange D., Goeke M., Madon D., Stauffer A., Tempesti G., Durand S., Marchal P. and Nussbaum P., "FPPA: A Field-Programmable Processor Array with self-repair and self-reproducing properties", 4th Reconfigurable Architecture Workshop, 1997

[Man97b] Mange D., Stauffer A. and Tempesti G., "Self-Replicating and Self-Repairing Field-Programmable Processor Arrays (FPPAs) with Universal Computation", Workshop on Evolvable Systems, IJCAI-97, Aug. 1997

[Man97c] Mange D., Madon D., Stauffer A. and Tempesti G., "Von Neumann Revisited: A Turing Machine with Self-Repair and Self-Reproduction Properties", Robotics and Autonomous Systems, Vol.22-1, 1997, pp.35-38

[Man98a] Mange D. and Tomassini M. (Eds.), Bio-Inspired Computing Machines, Presses Polytechniques et Universitaires Romandes, Switzerland, 1998

[Man98b] Mange D., Sanchez E., Stauffer A., Tempesti G. and Marchal P., "Embryonics: A New Methodology for Designing Field-Programmable Gate Arrays with Self-Repair and Self-Replicating Properties", IEEE Tr. On VLSI, Vol.6-3, September, 1999, pp.387-399

[Mnd97] Manderick B., "Evolvable Hardware: An outlook", in Higuchi, Iwata and Liu (eds.), Evolvable Systems: From Biology to Hardware, LNCS 1259, Springer-Verlag, 1997, pp.305-311

[Mar84] Margolus N., "Physics-like Models of Computation", Physica 10D, pp.81-95, 1984

[Mar96] Marchal P., Nussbaum P., Piguet D., Durand S., Mange D., Sanchez E., Stauffer A. and Tempesti G., "Embryonics: The birth of synthetic life", in Sanchez E. and Tomassini M. (Eds.), Towards Evolvable Hardware: The evolutionary engineering approach, LNCS 1062, Springer-Verlag, 1996, pp. 166-196

[Mar99] Marchal P., "Field-Programmable Gate Arrays", Communications of the ACM, Vol.42-4, April 1999, pp.57-59

[Mar99b] Marchal P., Invited presentation at the 1st NASA/DoD Workshop on Evolvable Hardware, Pasadena, CA, USA, July 1999, http://cism.jpl.nasa.gov/events/nasa_eh/papers/Presentation2.ppt

[Maz95] Mazzaferri R. and Murray T., "The Connection Network Class for Fault Tolerant Meshes", IEEE Trans. on Computers, Vol.44-1, January 1995, pp.131-138

[Mcc90] McCluskey E., "Design Techniques for Testable Embedded Error Checkers", IEEE Computer Vol.23-7, IEEE Computer Society, July 1990, pp.84-88

[Mei91] Meinhardt H., "Mechanisms of Biological Pattern Formation", in Peliti L. (Ed.), Biologically Inspired Physics, Plenum, 1991, pp.279-293

[Mil86] MIL-HDBK-217E, Reliability Prediction of Electronic Equipment, Military Handbook, United States Department of Defense, October, 1986

[Mis92] Misra K., Reliability Analysis and Prediction, Elsevier, 1992

[Moo91] Moore W. and Luk W. (Eds.), FPGAs, Abingdon EE&CS Books, 1991

[Mor82] Moret B., "Decision Trees and Diagrams", Computing Surveys, Vol.14-4, Dec.1982, pp.593-623

[Mor97] Morita K., "Logical Universality and Self-Reproduction in Reversible Cellular Automata", in Higuchi, Iwata and Liu (eds.), Proceedings of 1st International Conference on Evolvable Systems: From Biology to Hardware (ICES96), LNCS 1259, Springer-Verlag, 1997, pp.152-166

[Mori97] Morishita T. and Teramoto I., "Architecture of Cell Array Neuro-Processor", in Higuchi, Iwata and Liu (eds.), Proceedings of 1st International Conference on Evolvable Systems: From Biology to Hardware (ICES96), LNCS 1259, Springer-Verlag, 1997, pp.277-288

[Mur89] Murrell J.C. and Roberts L.M. (Eds.): Understanding Genetic Engineering, Ellis Horwood, Great Britain, 1989

[Mur97] Murakawa M., Yoshizawa S. and Higuchi T., "Adaptive Equalization of Digital Communication Channels Using Evolvable Hardware", in Higuchi, Iwata and Liu (eds.), Proceedings of 1st International Conference on Evolvable Systems: From Biology to Hardware (ICES96), LNCS 1259, Springer-Verlag, 1997, pp.379-389

[Nai97] Naito T., Odagiri R., Matsunaga Y., Tanifuji M. and Murase K., "Genetic Evolution of a Logic Circuit which Controls an Autonomous Mobile Robot", in Higuchi, Iwata and Liu (eds.), Proceedings of 1st International Conference on Evolvable Systems: From Biology to Hardware (ICES96), LNCS 1259, Springer-Verlag, 1997, pp.210-219

[Neg86] Negrini R., Sami M. and Stefanelli R., "Fault Tolerance Techniques for Array Structures Used in Supercomputing", IEEE Computer, Vol.19-10, October 1986, pp.78-87

[Neg89] Negrini R., Sami M. and Stefanelli R., "Fault Tolerance Through Reconfiguration in VLSI and WSI Arrays", MIT Press, Cambridge, 1989

[Nel90] Nelson V., "Fault-Tolerant Computing: Fundamental concepts", IEEE Computer, Vol.23-7, IEEE Computer Society, July 1990, pp.19-25

[Neu45] Von Neumann J., "First Draft of a Report on the EDVAC", 1945, in Randel B. (Ed.), The Origins of Digital Computers, 3rd ed., Springer-Verlag, 1982, pp.383-397

[Neu56] Von Neumann J., "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components", Automata Studies, Shannon C. and McCarthy J. (eds.), Annals of Math Studies, Num.34, Princeton Univ. Press, 1956, pp.43-98

[Neu66] Von Neumann J., Theory of Self-Reproducing Automata, edited and completed by Burks A., Univ. of Illinois Press, 1966

[NTT85] Nippon Telegraph and Telephone Corporation, Standard Reliability Table for Semiconductor Devices, March 1985

[Nus96] Nüsslein-Volhard C.: "Gradients that Organize Embryo Development", Scientific American, Vol.275-2, August, 1996, pp.38-43

[Nus97] Nussbaum P., Marchal P. and Piguet C., "Functional Organisms Growing in Silicon", in Higuchi, Iwata and Liu (eds.), Proceedings of 1st International Conference on Evolvable Systems: From Biology to Hardware (ICES96), LNCS 1259, Springer-Verlag, 1997, pp.139-151

[Ort97a] Ortega C. and Tyrrell A., "Fault-tolerant Systems: The way Biology does it!", Proceedings Euromicro 97 (Short Contributions), Budapest, IEEE CS Press, September, 1997, pp.146-151

[Ort97b] Ortega C. and Tyrrell A., " Biologically Inspired Reconfigurable Hardware for Dependable Applications", IEE Colloquium on Hardware Systems for Dependable Applications, London, November, 1997, Digest No: 97/335

[Ort98a] Ortega C. and Tyrrell A., "Design of a Basic Cell to Construct Embryonic Arrays", IEE Transactions on Computers and Digital Techniques, Vol.145-3, May, 1998, pp.242-248

[Ort98b] Ortega C. and Tyrrell A., "Evolvable Hardware for Fault-Tolerant Applications", IEE Colloquium on Evolvable Hardware Systems, London, March, 1998, Digest No: 98/233

[Ort98c] Ortega C. and Tyrrell A., "Biologically Inspired Real-Time Reconfiguration Technique for Processor Arrays", Proceedings of 5th IFAC Workshop on Algorithms and Architectures for Real-Time Control, Can-Cun, Mexico, Elsevier Science Ltd., Oxford, April, 1998, pp. 253-257

[Ort98d] Ortega C. and Tyrrell A., "MUXTREE revisited: Embryonics as a Reconfiguration Strategy in Fault-Tolerant Processor Arrays", Proceedings of ICES98, Lausanne, Switzerland, September, 1998, Lecture Notes in Computer Science 1478, Springer-Verlag, 1998, pp.206-217

[Ort99a] Ortega C. and Tyrrell A., "Biologically Inspired Fault-Tolerant Architectures for Real-Time Control Applications", Control Engineering Practice, July 1999, pp. 673-678

[Ort99b] Ortega C. and Tyrrell A., "Reliability Analysis in Self-Repairing Embryonic Systems", in Stoica A., Keymeulen D. and Lohn J. (Eds.), Procs. of 1st NASA/DoD Workshop on Evolvable Hardware, Pasadena, CA, IEEE Computer Society, July 1999, pp.120-128

[Ort99c] Ortega C. and Tyrrell A., "Reliability Analysis of Self-Repairing Bio-Inspired Cellular Hardware", IEE Colloquium on Evolutionary Hardware Systems, London, June 1999, Digest No: 99/033

[Ort99d] Ortega C. and Tyrrell A., "Self-Repairing Multicellular Hardware: A Reliability Analysis", in Floreano D., Nicoud J. and Mondada F. (Eds.), Advances in Artificial Life, Procs. of the 5th European Conference, ECAL'99, Lausanne, Switzerland, September 1999, Lecture Notes in Artificial Intelligence 1674, pp.442-446

[Oxf90] The Oxford Dictionary of Current English, Oxford University Press, 1990

[Pas95] Pasavento U., "An Implementation of von Neumann's Self-Reproducing Machine", Artificial Life 2, 1995, pp.337-354

[Pat95] Patterson D., "Microprocessors in 2020", Scientific American, September 1995, pp.48-51

[Pau96] Paul E. and Cox G., Beyond Humanity: CyberEvolution and future minds, Charles River Media, 1996

[Pel91] Peliti L. (Ed.), Biologically Inspired Physics, Plenum, 1991

[Per96] Perrier J., Sipper M. and Zahnd J., "Toward a Viable, Self-Reproducing Universal Computer", Physica D, No. 97, 1996, pp.335-352

[Pig99] Piguet C., Cicatrisation et Tolerance aux Pannes de Circuits Integres, Course on Cellular Arrays, EPFL, Lausanne, Switzerland, May 31, 1999

[Pin80] Pinsker H. and Willis W. (Eds.), Information Processing in the Nervous System, Raven Press, 1980

[Pol96] Poledna S., "Fault-Tolerant Real-Time Systems: The problem of Replica Determinism", Kluwer Academic Publishers, 1996

[Pom93] Pomeranz I., "Testing of Fault-Tolerant Hardware through Partial Control of Inputs", IEEE Trans. on Computers, Vol.42-10, Oct. 1993

[Pre84] Preston K. and Duff M., Modern Cellular Automata: Theory and applications, Plenum Press, 1984

[Pro48] Proceedings of the First Symposium on Large-Scale Digital Calculating Machinery, Harvard University Press, 1948

[Ran82] Randel B. (Ed.), The Origins of Digital Computers, 3rd ed., Springer-Verlag, 1982

[Rav97] Ravishankar I., "Foolproof and Incapable of Error?: Reliable Computing and Fault Tolerance, in Stork D. (Ed.), Hal's Legacy: 2001's Computer as Dream and Reality, MIT Press, 1997, pp.53-73

[Reg93] Reggia J., Armentrout S., Chou H. and Peng Y., "Simple Systems that Exhibit Self-directed Replication", Science, Vol.259, Feb.1993

[Ren98] Renovell M., Portal J. and Figueras J., "Testing the Interconnect of RAM-Based FPGAs", IEEE Design and Test of Computers, January-March 1998, pp.45-50

[Res94] Resnick Mitchel, Turtles, Termites and Traffic Jams, The MIT Press, 1994

[Roc98] Rocha L., "Evolutionary Systems and Artificial Life", Lecture Notes, http://www.c3.lanl.gov/~rocha/alife.html

[Ros83] Rosenberg A., "The Diogenes Approach to Testable Fault-Tolerant Arrays of Processors", IEEE Trans. on Computers, Vol.32-10, October 1983, pp.902-909

[Sak97] Sakanashi H., "Evolution of Binary Decision Diagrams for Digital Circuit Design using Genetic Programming", in Higuchi, Iwata and Liu (eds.), Proceedings of 1st International Conference on Evolvable Systems: From Biology to Hardware (ICES96), LNCS 1259, Springer-Verlag, 1997, pp.470-481

[Sam76] Sampson J., Adaptive Information Processing: An Introductory Survey, Springer-Verlag, 1976

[San96a] Sanchez E. and Tomassini M. (Eds.), Towards Evolvable Hardware: The evolutionary engineering approach, LNCS 1062, Springer-Verlag, 1996

[San96b] Sanchez E., "Field Programmable Gate Array (FPGA) Circuits", in Sanchez E. and Tomassini M. (Eds.), Towards Evolvable Hardware: The evolutionary engineering approach, LNCS 1062, Springer-Verlag, 1996, pp.1-18

[San97] Sanchez E., Mange D., Sipper M., Tomassini M., Pérez-Uribe A. and Stauffer A., "Phylogeny, Ontogeny and Epigenesis: Three sources of biological inspiration for softening hardware", in Higuchi, Iwata and Liu (eds.), Proceedings of 1st International Conference on Evolvable Systems: From Biology to Hardware (ICES96), LNCS 1259, Springer-Verlag, 1997, pp.35-54

[Sco95] Scott S., Samal A. and Seth S., "HGA: A Hardware-Based Genetic Algorithm", Procs. of the 1995 ACM/SIGDA 3rd. Int. Symposium on FPGAs, pp.53-59

[Sie86] Siemens Standard, SN29500- Reliability and Quality Specification Failure Rates of Components, 1986

[Sie90] Siewiorek D., "Fault Tolerance in Comercial Computers", IEEE Computer, Vol.23-7, July 1990, pp.26-37

[Sig89] Signorini J., "How a SIMD machine can implement a complex CA? A case study: Von Neumann's 29-state cellular automaton", Procs. of supercomputing '89 conference, Nov.1989

[Sim94a] Sims K., "Evolving Virtual Creatures", Computer Graphics, Annual Conference Series, Procs. SIGGRAPH '94, July 1994, pp.15-22.

[Sim94b] Sims K., " Evolving 3D Morphology and Behavior by Competition", Procs. Artificial Life IV, MIT Press, 1994, pp.28-39

[Sin88] Singh A., "Interstitial Redundancy: An Area Efficient Fault Tolerance Scheme for Large Area VLSI Processor Arrays", IEEE Trans. on Computers, Vol.37-11, November 1988, pp.1398-1410

[Sip95a] Sipper M., "An Introduction to Artificial Life", AI Expert, Sept. 1995, pp.4-8

[Sip95b] Sipper M., "Quasi-uniform Computation-universal Cellular Automata", in Morán F. (ed.), ECAL'95: 3rd European Conference on Artificial Life, LNCS 929, Springer-Verlag, 1995, pp.544-554

[Sip97a] Sipper M., "Designing Evolware by Cellular Programming", in Higuchi, Iwata and Liu (eds.), Proceedings of 1st International Conference on Evolvable Systems: From Biology to Hardware (ICES96), LNCS 1259, Springer-Verlag, 1997, pp.81-95

[Sip97b] Sipper M., Sanchez E., Mange D., Tomassini M., Perez-Uribe A. and Stauffer A., "A Phylogenetic, Ontogenetic and Epigenetic View of Bio-Inspired Hardware Systems",

IEEE Transactions on Evolutionary Computation, Vol.1-1, April 1997, pp.83-97

[Sip98] Sipper M., Mange D. and Perez-Uribe A. (Eds), Evolvable Systems: From Biology to Hardware, Proceedings of ICES98, Lausanne, Switzerland, September, 1998

[Sip99] Sipper M., Mange D. and Sanchez E., "Quo Vadis Evolvable Hardware?", Communications of the ACM, Vol.42-4, April 1999, pp.50-56

[Som97] Somani A. and Vaidya N., "Understanding Fault Tolerance and Reliability", IEEE Computer, Vol.30-4, April 1997, pp.45-50

[Ste97] Stewart D., "Cellular Automata: an interactive essay", http://www.foresight.co.uk/stewart/alife

[Sti94] Stiffler J., "Fault-Tolerant Architectures: Past, present and (?) future", in [Ban94]

[Sto89] Stone R., "Reliable Computing Systems: A review", University of York, Dept. of Computer Science, YCS 110, 1989

[Sto93] Stone H., High-Performance Computer Architecture, Addison-Wesley, 1993

[Sto97] Stork D. (Ed.), Hal's Legacy: 2001's Computer as Dream and Reality, MIT Press, 1997

[Sto99] Stoica A., Keymeulen D. and Lohn J. (Eds.), Procs. of 1st NASA/DoD Workshop on Evolvable Hardware, Pasadena, CA, USA, IEEE Computer Society, July 1999

[Sut89] Sutherland I., "Micropipelines", Communications of the ACM, Vol.32-6, June 1989, pp.720-738

[Tab76] Tabloski T. and Mowle F., "A Numerical Expansion Technique and Its Application to Minimal Multiplexer Logic Circuits", IEEE Trans. on Computers, Vol.25-7, July 1976, pp.684-702

[Tas77] Tasar O. and Tasar V., "A Study of Intermitent Faults in Digital Computers", AFIPS Conference Proceedings, 1977, pp.807-811

[Tay95] Taylor C. and Jefferson D., " Artificial Life as a Tool for Biological Inquiry", in Langton C. (ed.), Artificial Life: an Overview, MIT Press, 1995, pp.30-45

[Tem94] Tempesti G., Mange D. and Stauffer A., "A Self-repairing FPGA Inspired by Biology", Internal report, Logic Systems Laboratory, EPFL, 1994

[Tem95] Tempesti G., "A New Self-reproducing Cellular Automaton Capable of Construction and Computation", in Morán F. (Ed.), ECAL'95: 3rd European Conference on Artificial Life, LNCS 929, Springer-Verlag, 1995

[Tem97] Tempesti G., Mange D. and Stauffer A., "A Robust Multiplexer-based FPGA Inspired by Biological Systems", Special Issue of Journal of Systems Architecture on Dependable Parallel Computer Systems, February 1997, pp.719-733

[The68] Theobald D., An Introduction to the Philosophy of Science, Methuen and Co Ltd, 1968

[Tho95] Thompson A., "Evolving Fault Tolerant Systems", First IEE/IEEE Intl. Conference on Genetic Algorithms in Engineering Systems (GALESIA '95), Sept. 1995

[Tho96a] Thompson A., "Unconstrained Evolution and Hard Consequences", in Sanchez E. and Tomassini M. (Eds.), Towards Evolvable Hardware: The evolutionary engineering approach, LNCS 1062, Springer-Verlag, 1996, pp.136-165

[Tho96b] Thompson A., "Silicon Evolution", in Koza J. (Ed.), Proceedings of Genetic Programming 1996, MIT Press, 1996, pp.444-452

[Tho96c] Thompson A., "Evolutionary Techniques for Fault Tolerance", IEE Proceedings of Intl. Conference on Control (CONTROL '96), 1996.

[Tho97] Thompson A., "An Evolved Circuit, Intrinsic in Silicon, Entwined with Physics", in Higuchi, Iwata and Liu (eds.), Proceedings of 1st International Conference on Evolvable Systems: From Biology to Hardware (ICES96), LNCS 1259, Springer-Verlag, 1997,

pp.390-405

[Tho99] Thompson A. and Layzell P., "Analysis of Unconventional Evolved Circuits", Communications of the ACM, Vol.42-4, April 1999, pp.71-79

[Tis99] Tisserand A., Marchal P. and Piguet C., "An On-Line Arithmetic Based FPGA for Low Power Custom Computing", Procs. 7th Annual IEEE Symposium on Field Programmable Custom Computing Machines, Napa CA, 1999

[Tof87] Toffoli T. and Margolus N., Cellular Automata Machines: A new environment for modelling, MIT press, 1987

[Tom96] Tomassini M., "Evolutionary Algorithms", in Sanchez E. and Tomassini M. (Eds.), Towards Evolvable Hardware: The evolutionary engineering approach, LNCS 1062, Springer-Verlag, 1996, pp. 19-47

[Tur90] Turino J., Design to Test: A definitive Guide for Electronic Design, Manufacture and Service, Van Nostrand Reinhold, New York, 1990

[Tyr94] Tyrrell, A.M. 'Evaluation of Fault Tolerant Structures on a Transputer Module', Proceedings of 2nd Euromicro Workshop on Parallel and Distributed Processing, Malaga, IEEE Computer Society Press, pp 134 - 140, January 1994

[Tyr99] Tyrrell A. "Computer Know Thy Self!? A Biological Way to Look at Faul Tolerance, 2nd IEE/EUROMICRO Workshop on Dependable Computing Systems, Milan, September, 1999

[Vil97] Villaseñor J. and Mangione-Smith W., "Configurable Computing", Scientific American, June 1997, pp.54-49

[Vol91] Volkenstein M., "Physical Approaches to Biological Evolution", in Peliti L. (Ed.), Biologically Inspired Physics, Plenum, 1991, pp.301-315

[Wak78] Wakerly J., Error-detecting codes, self-checking circuits and applications, Thomond Books, 1978

[Wat70] Watson J., The Double Helix, Penguin Books, 1970

[Wat92] Watson G., "MIL Reliability: A new approach", IEEE Spectrum, August, 1992, pp.46-49

[Wei77] Weide B., "A Survey of Analysis Techniques for Discrete Algorithms", ACM computing surveys, Vol.9-4, Dec. 1977, pp.291-313

[Wei91] Weisbuch G., "Problems in Theoretical Immunology", in Peliti L. (Ed.), Biologically Inspired Physics, Plenum, 1991, pp.249-262

[Wil86] Williams T., "Design for Testability", in Antogneti P., Pederson D. and Mann de H. (Eds.), Computer Design Aids for VLSI Circuits, Martinus Nijhoff, 1986, pp.359-416

[Wol74] Wolpert L., The Development of Pattern and Form in Animals, Oxford University Press, 1974

[Wol83] Wolfram S., Cellular Automata (1983), in [Wol97]

[Wol86] Wolfram S., Theory and Applications of Cellular Automata, World Scientific, 1986

[Wol91] Wolpert L., The Triumph of the Embryo, Oxford University Press, 1991

[Wol97] Wolfram S., Publications on Cellular Automata, www.wolfram.com/s.wolfram/articles

[Wol98] Wolpert L., Beddington R. and Brockes J., Principles of Development, Oxford Universtity Press, 1998

[Xan95] Xanthakis R., "Immune System and Fault-tolerant Computing", Evolution Artificielle 94, Cepadues cop., 1995

[Xil94] Xilinx, The Programmable Logic Data Book, Xilinx Inc., 1994

[Xil99] Xilinx, Virtex™ 2.5V Advance Product Specification, version 1.6, July 1999

[Yam97] Yamamoto J. and Anzai Y., "Autonomous Robot with Evolving Algorithm Based on Biological Systems", in Higuchi, Iwata and Liu (eds.), Proceedings of 1st International Conference on Evolvable Systems: From Biology to Hardware (ICES96), LNCS 1259, Springer-Verlag, 1997, pp.220-233

[Yao97] Yao X. and Higuchi T., "Promises and Challenges of Evolvable Hardware", in Higuchi, Iwata and Liu (eds.), Evolvable Systems: From Biology to Hardware, LNCS 1259, Springer-Verlag, 1997, pp.55-78

[Yao99a] Yao X. and Higuchi T., "Promises and Challenges of Evolvable Hardware", IEEE Trans.on Systems, Man, and Cybernetics- Part C: Applications and Reviews, Vol.29-1, Feb. 1999, pp.87-97

[Yao99b] Yao X., "Following the Path of Evolvable Hardware", Communications of the ACM, Vol.42-4, April 1999, pp.47-49

[Yau70] Yau S. and Tang C., "Universal Logic Modules and their Applications", IEEE Trans. on Computers, Feb. 1970, pp.141-149

[Yor93] York T., "Survey of Field Programmable Logic Devices", Microprocessors and Microsystems, Vol.17-7, Sept.1993, pp.371-381

[Yos72] Yost H., Cellular Physiology, Prentice-Hall, 1972

[Zeb97] Zebulum R., Pacheco M. and Vellasco M., "Evolvable Systems in Hardware Design: Taxonomy, Survey and Applications", in Higuchi, Iwata and Liu (eds.), Evolvable Systems: From Biology to Hardware, LNCS 1259, Springer-Verlag, 1997, pp.344-358

[Zif83] Rosenfield I, Ziff E. and Van Loon B., DNA For Beginners, Writers and Readers Pub., 1983