Journal of Systems Architecture: Special Issue on Dependable Parallel Computer Systems, 43(10), 1997.

A Robust Multiplexer-Based FPGA Inspired By Biological Systems

Gianluca TEMPESTI*, Daniel MANGE, André STAUFFER
Logic Systems Laboratory
Department of Computer Science
Swiss Federal Institute of Technology (EPFL)
CH-1015 Lausanne, Switzerland

ABSTRACT

Biological organisms are among the most robust systems known to man. Their robustness is based on a set of processes which cannot be adapted directly to the world of silicon, but can provide an inspiration for the design of robust circuits. This paper introduces a multiplexer-based FPGA which we made capable of self-test and self-repair using an approach loosely based on biological mechanisms at the cellular level. The system is designed to provide on-line self-test and self-repair using a completely distributed system and a minimal amount of additional logic.

1. INTRODUCTION

It is undeniable that some of the features of biological organisms (healing, growth, evolution, etc.) would be extremely beneficial if applied to electronic circuits. Of course, a direct transfer of the biological mechanisms to silicon is impossible, but the Embryonics project [5,6,13] tries to determine if, with the appropriate modifications, some of the concepts behind these mechanisms can be adapted to the design of logic circuits. In particular, we examine biological organisms at the cellular level.

One of the most interesting features of biological mechanisms at the cellular level is their ability to self-repair: cells are continuously killed and created but, throughout its "adult" life (that is, after the growth phase is over), an organism continues to function as if all of its original cells were still active. The basis of this robustness is the fact that each cell contains the description of the entire organism (the *genome*), and can therefore replace any other cell through simple self-reproduction. The concept of genome is in fact at the root of our approach to self-

* To whom correspondence should be addressed.
Phone: (+41 21) 693 2676
Fax: (+41 21) 693 3705
Email: tempesti@di.epfl.ch

repair. Unfortunately, the silicon substrate of our cells does not allow us to create and destroy cells with the necessary ease, which forces us to limit this kind of solution to the more extreme cases. Therefore, we need to make our cells more robust than their biological equivalent.

Since our cells are implemented, as we will see, using an FPGA (Field-Programmable Gate Array) of our own conception (called MUXTREE, Fig. 1) [6,7], this requirement implies that the FPGA itself be robust. We therefore conceived and implemented a self-repair mechanism at the FPGA level. This paper describes our FPGA and the additional features which we introduced to implement some of the pseudo-biological properties of Embryonics, and in particular self-repair and self-reproduction.

To understand the overall approach to the design of our system, it is therefore important to understand the basic philosophy of the Embryonics project. Therefore, the next section will briefly describe the salient points of the project, and will serve as background for the following sections, in which we will outline the main features of our system.

2. EMBRYONICS

Embryonics is essentially an *experiment*, in the sense that is a project conceived not so much to achieve a specific goal, but rather to look for insights by applying new concepts to a known field. In our case, we are trying to determine if interesting results can be obtained by applying biological concepts (i.e., concepts which are usually associated with biological processes) to computing, and in particular to the design of computer hardware.

Of course, carbon-based biology and silicon-based computing are different enough that no straightforward one-to-one relationship between biological and computing processes can be established. However, through careful interpretation, some basic biological *concepts* can indeed be adapted to circuit design, and some biological processes are indeed extremely interesting from a computer designer's point of view (clear examples of this are healing and evolution).

The first fundamental difference between the two fields lies in the material itself with which they are concerned. Biology deals with carbon-based structures which are routinely created and destroyed (e.g., cells). Computer science deals with silicon-based structures (circuits) which, conversely, can be neither created nor destroyed (or at least, not easily). This very basic difference, which would at first sight represent an insurmountable obstacle, can however be overcome by observing that, in reality, computer science does not deal with circuits but rather with *information*, which can indeed be created and destroyed. Thus information can be seen as the computing equivalent of biological structures, while the hardware can be seen as providing the *physics* of the system, that is, the immutable layer which provides the basic rules for the behavior of information.

A second difference between the two worlds is of course their dimensionality. The biological world operates in three dimensions, while (for the moment) circuits only operate in two. There is no immediate solution to this problem, but fortunately it does not represent a major inconvenience given the current state of the project (it does imply that the connectivity will at some point become a limiting factor, but it does not prevent the development of the basic concepts).

These are just two of the more obvious differences between the silicon-based world of computing and the carbon-based world of biology. The list is far from

exhaustive, but it should illustrate the difficulties of our attempt to bridge the gap between the two worlds. Clearly, the task is not trivial. However, we feel that such an attempt, imperfect as the results might be, could provide some interesting insights which could prove beneficial to the development of circuits.

The basic unit of biological life is the cell, defined as the smallest structure containing the description of the entire organism (i.e., the *genome*). The equivalent of biological cells in our system are relatively simple processors, based on a binary decision machine and a set of programmable connections [5,12]. Our system consists of a two-dimensional array of these cells, where each processor contains a copy of the same program, but executes only a specific portion of the code, depending on its coordinates within the array (in much the same way as a skin cell and a liver cell both contain the same genome, but interpret different parts of it according to their functionality, which is itself determined by the cell's spatial location). Since the coordinates are computed locally on the basis of information received from the cell's neighbors, the entire system can be seen as a SPMD (Single-Program Multiple-Data) system where the data stream is provided by the cell's neighbors.

This kind of structure lends robustness to our system, as a cell can be replaced by another simply through the recomputation of the coordinates. We are in the process of designing a system whereby a faulty cell simply becomes transparent to the array, thus causing a recomputation of the coordinates through the array. This, assuming "spare" cells are available, should allow the system to retain its functionality in the presence of faults.

This paper, however, does not deal directly with this part of the system, but rather with fault tolerance at a lower level. In fact, the cells are themselves implemented using an FPGA of our own design, called MUXTREE [6,7], based on a two-input multiplexer (Fig. 1). The use of an FPGA allows us a much greater flexibility in the design of cells than would be possible using custom-designed chips. In particular, it provides a uniform surface of logic gates which can be used to implement cells of any given size. We did in fact design a prototype cell with fixed dimensions, but we quickly realized that to do so would put an inherent limit to the size of a cell. The use of the MUXTREE layer, which was designed especially for this task, allows us to create arrays of cells of any given dimension. Moreover, since the MUXTREE elements are themselves capable of self-test and self-repair, we can exploit a two-level hierarchical self-repair scheme [11, 16, 18], in which smaller faults are detected an repaired at the MUXTREE level, while more important ones are left to the cellular level.

FPGAs thus enter the Embryonics project at a very basic stage. Their reprogrammability is a feature ideally suited to our needs, since it provides the possibility of directly accessing the hardware. The hardware itself (the silicon) is not modified, but its function can be, and since we are dealing with the processing of information, the hardware's function is indeed what we are interested in[1].

[1] Another feature which, at first sight, would seem to recommend the use of FPGAs in the Embryonics project is the fact that they consist of a two-dimensional array of *cells*. Unfortunately, the use of the words *cell* and *cellular* in this context is somewhat misleading and represents a recurring problem of terminology. As we mentioned, the defining feature of a biological cell is the presence of the genome: in biology, a cell is the smallest structure containing the description of the entire organism. In this sense, a FPGA cell cannot be considered the equivalent of a biological cell. A more appropriate analogy can probably be made with a *molecule*, and we will reserve the term *cell* for the

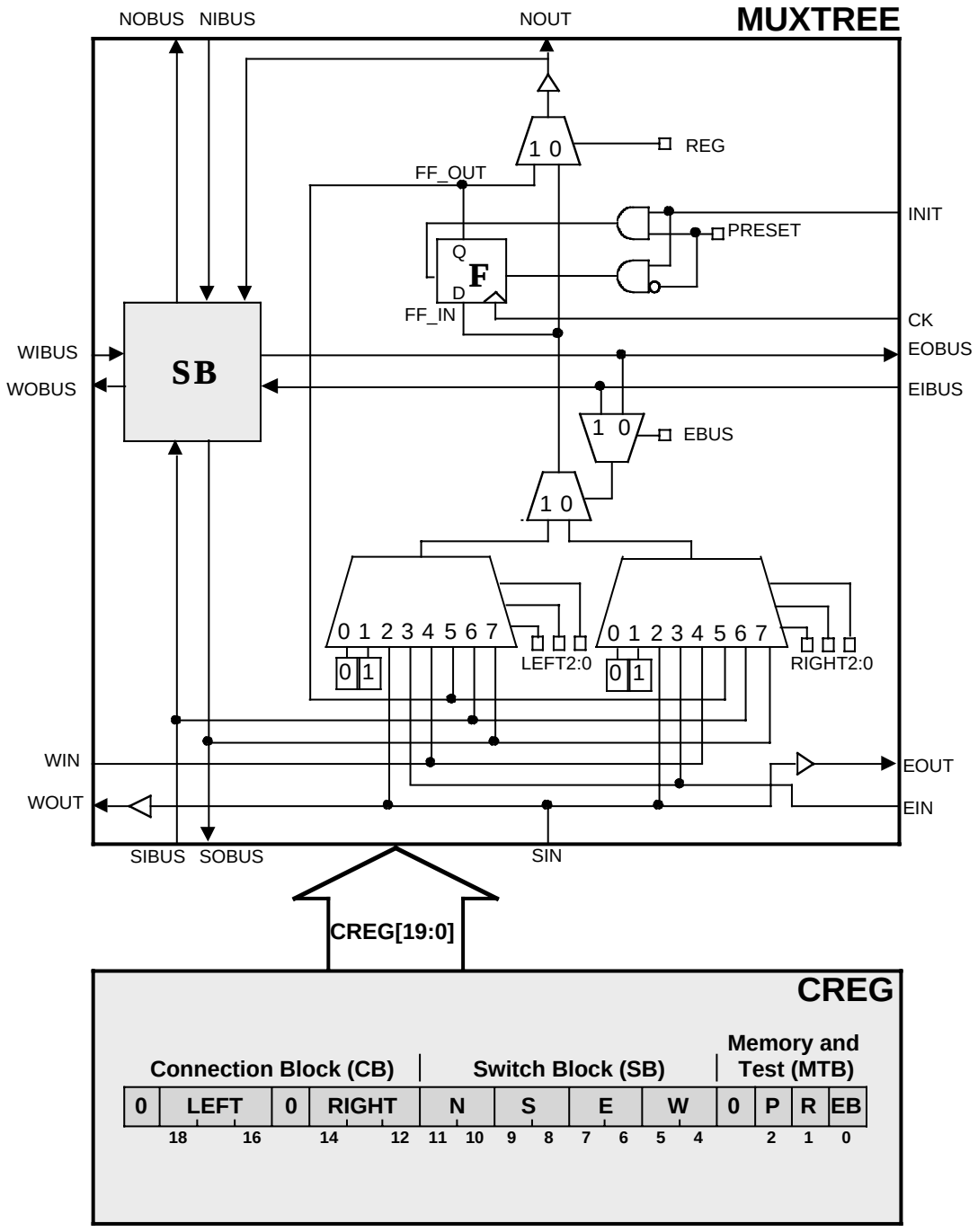

Fig. 1a: Logic layout of a MUXTREE element, including the configuration register.

more complex processing elements described above, which can be more accurately compared with biological cells, using the term *element* to refer to FPGA cells.

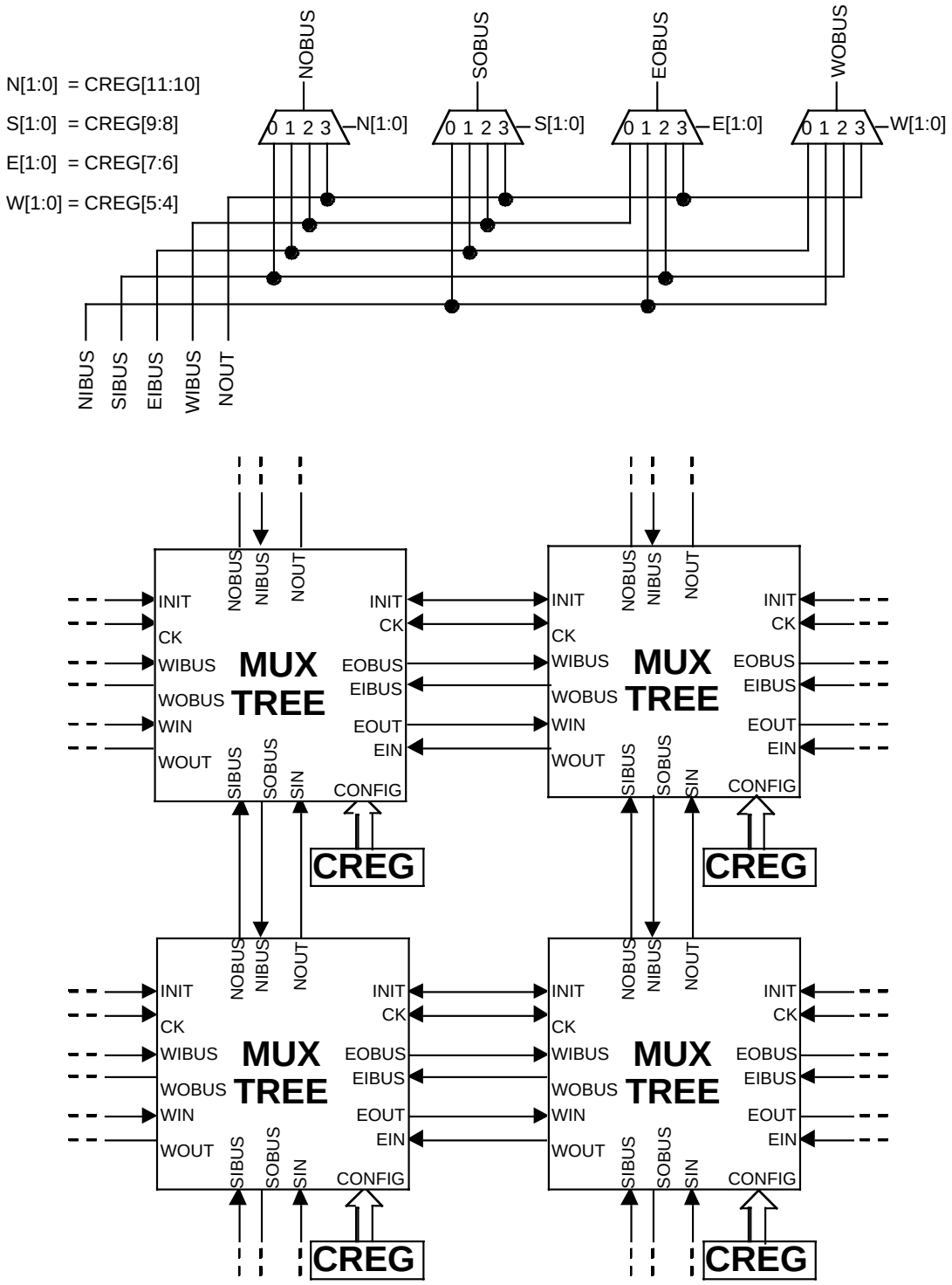

Fig. 1b: Logic layout of the switch block (top) and an example of a fully-connected MUXTREE array (bottom).

As we mentioned, biological systems achieve robustness through massive redundancy and by physically replacing faulty cells. This level of robustness is not really achievable in computing systems, where cells cannot be physically replaced and where redundancy is effectively limited by practical considerations. Given the simplicity of the basic MUXTREE elements and the complexity of the cells, a single cell must necessarily use a relatively large number of cells. Increasing the size of a molecule (by increasing the complexity of the MUXTREE elements) would probably ameliorate this requirement, but definitely not eliminate it. It is therefore clear that entire cells should not be "killed" unless absolutely necessary, which in turn implies that some robustness is needed at the MUXTREE level. To this end, we have added self-test and self-repair capabilities to the basic MUXTREE element. The next sections will describe the system we have developed to implement these features.

3. THE SELF-TEST MECHANISM

3.1 Introduction

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

Generally, the goal of self-test is the detection of defects occurring in the system. The standard modelization of these defects is as *stuck-at faults*. That is, the only defects which are actually considered are those which have the net effect of fixing the value of a line to 0 (*stuck-at-0 faults*) or 1 (*stuck-at-1 faults*). This definition by no means covers all possible electrical faults in a circuit (for example, a physical fault could cause lines to be left floating, that is, in a high-impedance state), but it does cover the most common, and the great majority of the research in the area of testing is concerned with this kind of faults [1, 8, 10].

The first added constraint is imposed by the need for self-repair: any self-repair mechanism will be able to know not only that a fault has occurred somewhere in the circuit, but also exactly where. Thus, our system has to be able to perform not only *fault detection*, but also *fault location*.

A second constraint is that we desire our system to be completely distributed. As a consequence, the self-test logic must be distributed among the MUXTREE elements, and the use of a centralized system (fairly common in self-test systems [8, 9, 11]) is therefore not a viable option. This constraint is due to our attempt to model biological mechanisms (organisms do indeed have centralized organs and mechanisms to detect defects, but they act at a much higher level).

The parallel with biological systems also imposed a third constraint: that the self-test occur on-line [1]. Organisms monitor themselves constantly, in parallel with their other activities, and therefore our FPGA should be able to test itself while operating. As we will see, this constraint proved itself too strong to be truly feasible, and was somewhat relaxed in our current implementation.

Finally, the relatively small size of the MUXTREE elements imposed a constraint on the amount of logic allowed for self-testing. The minimization of the logic was not really our main goal, and we did indeed trade logic in favor of satisfying our other constraints. However, we made an attempt to keep the testing logic down to a reasonable size, which imposed some restrictions on the design of the system.

In conclusion, we tried our best to respect the many external constraints, some of which contradictory, when designing our self-test mechanism. These constraints excluded the use of most of the "standard" approaches to self-testing (output pattern analysis, parity checking, redundant coding, etc. [1, 14]). The result is a system which is not perfect in any one respect, but provides what we feel is a reasonably well-balanced compromise.

3.2 Overview

A system capable of performing on-line self-test necessarily implies the presence of some form of redundancy. In fact, for the system to be able to operate on-line, the logic necessary for its operation cannot be used also to perform the test. Thus, additional logic is necessary and the logic test can only work by comparison between two or more values.

A careful examination of the MUXTREE element (Fig. 1) reveals that it can be divided into three separate parts, which will be handled separately by the test system: the configuration register (REG), the connections between elements, and the functional part of the element (which includes, essentially, all the logic in Fig. 1a apart from the configuration register). The amount of logic required by each of the three parts is unequal, with the configuration register taking up by far the largest number of transistors.

For each of these components, we developed a system capable of meeting our requirements in the measure permitted by the constraints imposed by the circuit. We will now examine more in detail our current solutions for the test of each of the three parts of the MUXTREE element.

3.3 The Functional Part

There are essentially two ways for redundancy to be implemented for the functional part of the element: in time or in space (Fig. 2) [8]. After experimenting with time redundancy in an attempt to reduce the amount of testing logic, we finally settled for a mechanism which exploits space redundancy. This approach provides a much greater simplicity of design and operation (particularly where fault locations is concerned) at the expense of a surprisingly small additional amount of logic.

The approach is based on double redundancy (Fig. 3): the functional part of the element is duplicated and the outputs NOUT of the two copies are compared. If a difference is detected, the element is faulty and the self-repair mechanism (see below) is activated.

There is one exception to the double redundancy approach, and it concerns the flip-flop contained within the element. In fact, where self-test is concerned, two copies of the flip-flop are sufficient to assure that faults will be detected. However, self-repair requires that the state of the circuit not be lost when a fault occurs. A fault occurring in one of only two copies of the flip-flop would not allow the state of the circuit to be maintained, since it would not be possible to determine which of the two values is correct. To be able to retain the state of the circuit, it is thus necessary to introduce a third flip-flop (D3), which will allow us to determine the correct value to use in the self-repair phase simply by computing the majority function of the three flip-flops. Of course, a comparison between the inputs of the two original flip-flops is also required, to ensure that the third flip-flop does not receive an incorrect value.

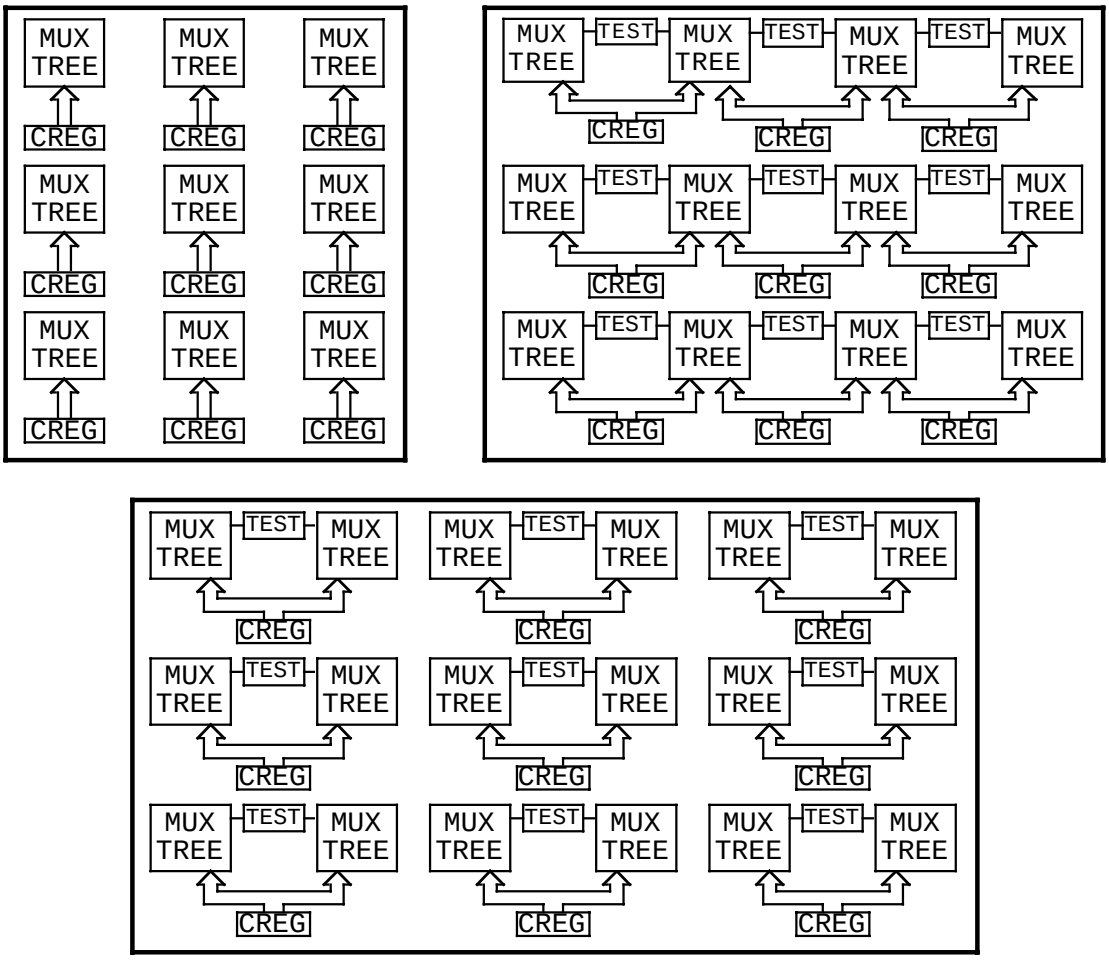

Fig. 2: Application of redundancy to a MUXTREE array: no redundancy (top left), time redundancy (top right), space redundancy (bottom).

This system is extremely simple in conception, and the additional amount of logic is not quite as important as one might believe, considering that the surface of silicon used by the functional part of the element is relatively small compared with that of the configuration register (an early VLSI version of a MUXTREE array revealed that the configuration register occupied over 80% of a cell's surface).

3.4 The Connections

Testing the connections is a fundamental part of our system, given that a network of MUXTREE elements is a very connection-heavy circuit. Unfortunately, the task is far from simple [8,14].

The first observation which can be made in considering the problem of testing connections is that, unfortunately, the only way to test a connection (which is, after all, a line) is by duplication. Thus, if we want to test all the connections, it is necessary to duplicate all the lines, an expensive (in terms of additional silicon) proposition, especially since all the connections will have to be redirected for self-repair (see below).

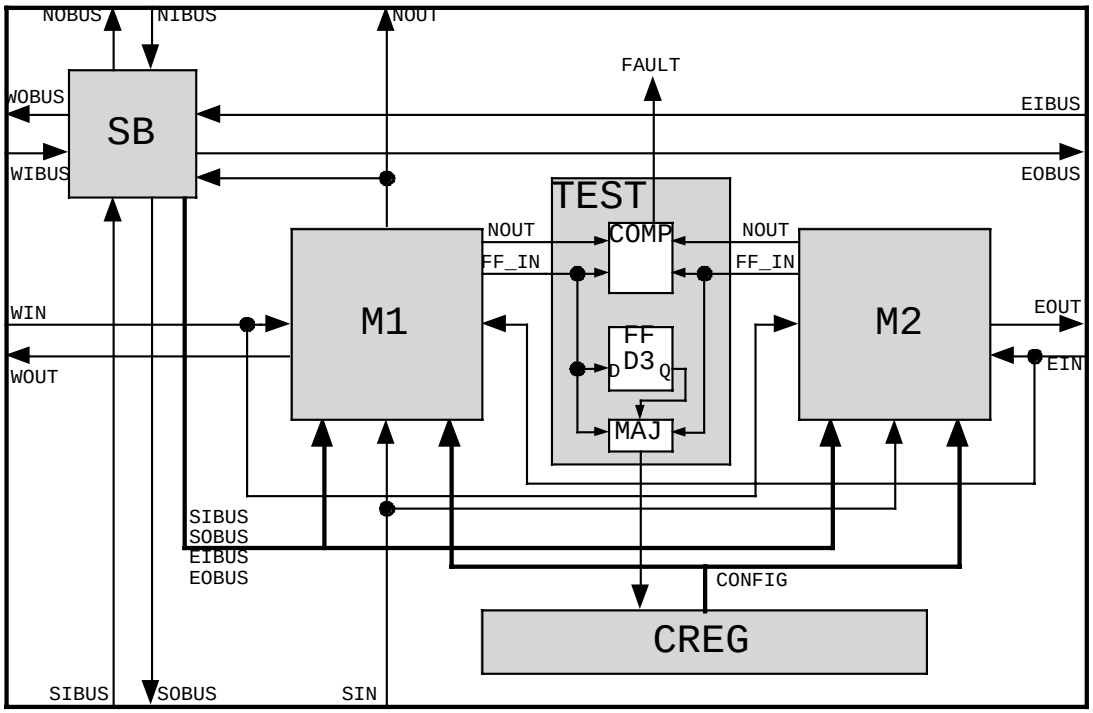

Fig. 3: Layout of a MUXTREE element using space redundancy for self-testing.

A second remark concerns the phenomenon of *fault propagation*: if the two copies of the functional part of the element are connected by separate connection networks, then the two networks will propagate different values and different outputs will be detected not only in the faulty element, but also in the elements connected to the faulty one through combinational logic (fault propagation is stopped by an active flip-flop). It is thus necessary to distinguish between the faulty element and those elements to which the fault is only propagated. This can be accomplished by extending the comparison to all the inputs (as well as the output) of the two copies of the functional part of the element. Obviously, only those elements where a difference is detected on the output but not on the inputs are actually faulty. The problem can thus be solved, but only using a relatively large additional amount of logic. Moreover, this approach allows us to detect that a fault has indeed occurred in the connections, but not exactly where, which causes problems where self-repair is concerned.

As a consequence of these observations, it should be apparent that the test of the connections is in fact one of the most sensitive areas of the system, and we are still evaluating the possible options to find the most efficient solution.

3.5 The Configuration Register

Testing the configuration register implies an entirely different set of problems. Duplicating the registers, the simplest way to perform the test, is not an option, since they occupy by far the largest amount of silicon in the element. One observation, however, somewhat simplifies the problem. The observation is that, in fact, the contents of the configuration register are not supposed to vary during the operation of the network. This implies that, should one of the bits change of value while the circuit is operating, we can conclude that a fault has occurred.

This observation simplifies the task of detecting a fault in the register, but, unfortunately, not as much as we could hope. In fact, the logic necessary to detect that the value of a bit has changed is not at all negligible (n-1 XOR gates for a n-bit register). This fact, as well as the consideration that such a fault cannot be repaired (since no duplicate of the information is available), has led us to decide not to test the register on-line, but rather limit the test to the configuration phase (i.e., the moment when the configuration is being charged into the registers).

In fact, there exist some relatively simple and inexpensive (in terms of additional logic) solutions to the problem of testing the register as it is being configured. One such solution is based on the observation that the configuration register is in fact a shift register. Since a fault in a shift register is propagated along the chain, it is possible to define a "configuration" which allows us to determine whether a fault is present anywhere in the register. Charging this configuration into all the registers in parallel, *before* charging the actual configuration stream, will let us determine whether a fault is present in any of the elements, which will then be "killed" and replaced by another according to the mechanism described in the next section.

A configuration which satisfies these criteria is shown in Fig. 4. Assuming that all errors can be modeled as either stuck-at-0 or stuck-at-1 faults, this simple mechanism allows errors in the register to be detected as it is being filled. In fact, such a fault will cause the shift register to fill with either all 0s or all 1s starting from the location of the fault. In the presence of a fault, when the 11 sequence arrives at the tail of the register (on the left in the figure), the head (the two elements at the extreme right) will contain either two zeros or two ones, rather that the expected 01 sequence.

Such a configuration, shifted into the register, will therefore allow the few gates shown to detect a fault anywhere in the register, with the exception of the very first element (and, of course, of the connection itself). The test of the first element, however, occurs automatically, since the last bit of the configuration corresponds not to the last value to be stored into the configuration register, but rather to the value to be stored in the flip-flop contained in the functional part of the element (which can thus be initialized to the desired value). Since there are in fact three copies of this flip-flop (as mentioned above), the majority circuit already in place allows us to test it without additional logic (Fig. 4 bottom).

4. THE SELF-REPAIR MECHANISM

In conclusion, our self-test mechanism provides on-line fault detection for the functional part of the elements. It also provides full fault location for the functional part and, optionally, partial fault location for the connections. On the other hand, it provides neither on-line fault detection nor on-line fault location for faults occurring within the configuration register.

This is certainly an inconvenience where self-repair is concerned, but less so than might be expected. In fact, it should be apparent that a fault occurring on either the configuration register or the connections cannot really be repaired at this level, since the first represents an unrecoverable loss of data (the data stored in the register is not duplicated), while the second is not compatible with our self-repair mechanism which, as we will see, necessarily assumes that the connections are functioning correctly. These kinds of faults cannot thus be repaired at this level, and we will rely

on reconfiguration at the cellular level (which we mentioned above) to handle them by replacing the entire cell.

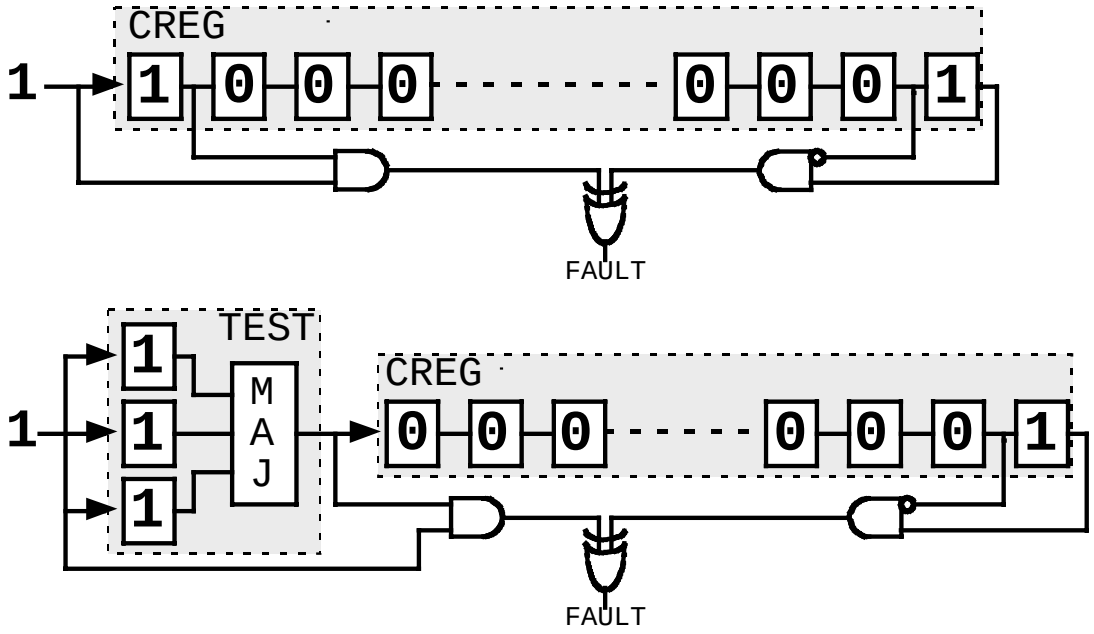

Fig. 4: A self-test sequence applied to the configuration register (top) and to the register-FF chain (bottom), including the fault-detection logic.

Faults occurring in the functional part of the element, however, do not cause any data loss (thanks also to the triplication of the flip-flop) or any problem concerning the connectivity of the network, and an attempt at repairing the damage can thus be effected.

The mechanism we propose to repair such faults relies on the reconfiguration of the network (Fig. 5) through the replacement of the faulty element by its right-hand neighbor (Fig. 6). Whenever a fault is detected, the FPGA effectively goes off-line for the time required for the element to be replaced (somewhat like an organism becomes incapacitated during an illness). Such replacement is not as trivial as in the case of an entire cell, which contains the configuration of every other cell within its genome, but requires the configuration of the faulty element to be shifted to it neighbor. Once the shift has occurred, the element “dies” with respect to the network, that is, the connections are rearranged to avoid the faulty element. This rearrangement is effected very simply by deviating the north/south connections to the right and by rendering the element transparent with respect to the east-west connections. Of course, the configuration of the neighbor needs to be itself shifted to the right, and so on until a *spare element* is reached, at which point the reconfiguration stops and the normal operation of the FPGA resumes.

The amount of logic required by this mechanism is not very important, particularly because the reconfiguration (that is, the shifting of the configuration) is limited to a single element and to a single direction. Of course, this limits the number of faults which can be successfully repaired to one per line between two columns of spare elements, but this limitation is compensated by the fact that the frequency of spare vs. used columns in the array is itself programmable (see section 5). The spare columns correspond in fact to a particular configuration of the MUXTREE element,

and as many spare columns as desired can be added when the configuration of the cell is generated. Of course, a large enough number of faults will still saturate the mechanism, but in this case the cell-level reconfiguration mechanism can take over and replace the entire cell (to this end, a special kill signal is generated by the MUXTREE elements when self-repair is no longer possible).

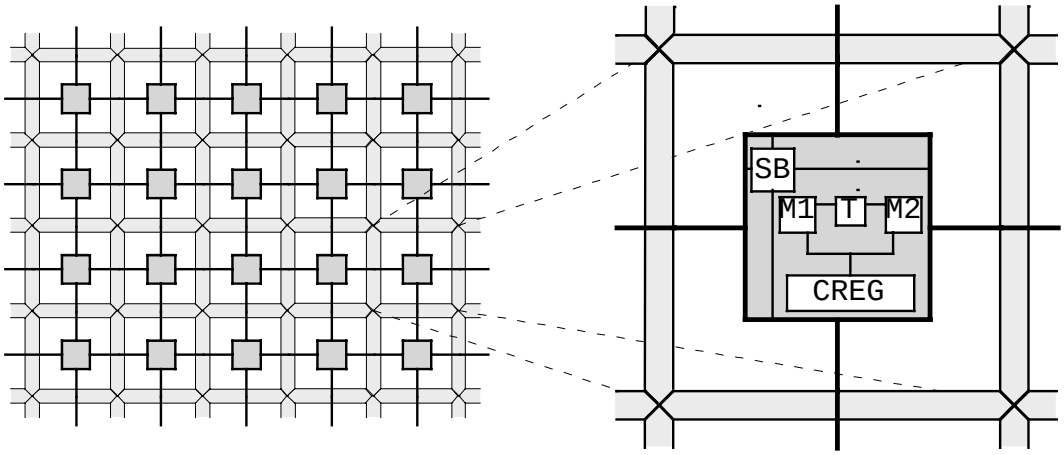

Fig. 5: An array of MUXTREE elements.

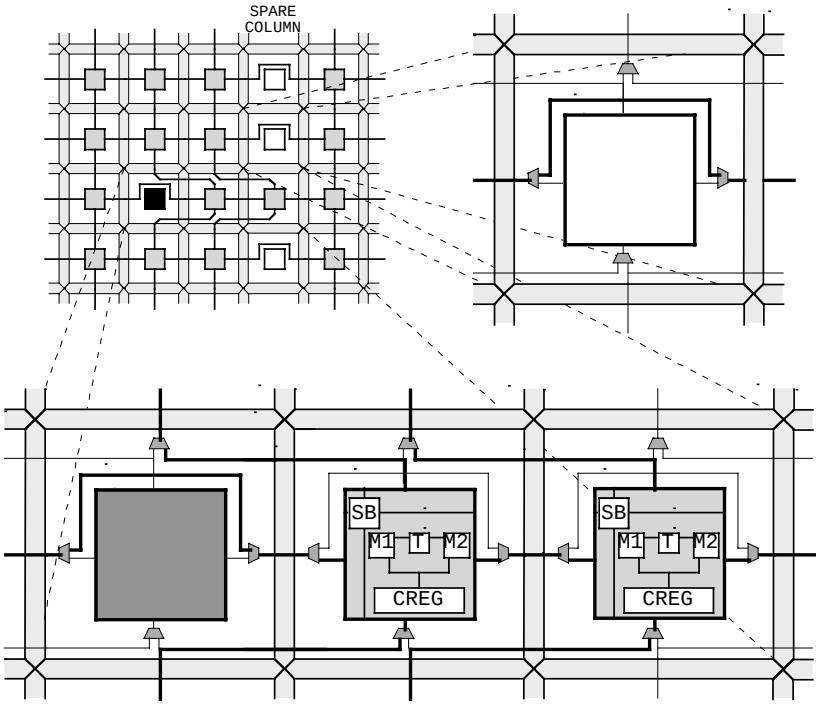

Fig. 6: The self-repair mechanism in a MUXTREE array with spare columns.

The last relevant feature of our self-repair mechanism that we will mention is one that would apparently seem a disadvantage, but reveals itself to be a very positive feature to a more careful examination. We are talking about the fact that the entire self-test and self-repair mechanism we have described is volatile: powering down the circuit results in the loss of all information regarding the condition of the circuit. As we said, this might seem a disadvantage, since it forces us to find and repair the same faults every time the circuit is reconfigured. However, it is crucial to consider that by far the largest part of the faults occurring within a circuit throughout its life are *temporary faults*, that is, faults which will "vanish" very shortly after having appeared [9,10]. It would therefore be very wasteful to permanently repair faults which are in fact only temporary, and the volatile nature of our mechanism turns out to be an important advantage.

5. THE CONFIGURATION MECHANISM

As we have repeatedly mentioned, our system is based on a network of identical cells, each containing a copy of the genome, that is, of the configuration of every cell in the network. This approach, which is obviously not optimal with respect to standard circuit design rules, does however provide us with a set of features which are interesting from the point of view of Embryonics, and in particular a certain inherent robustness which is a vital feature of biological systems. But another, less evident advantage of a truly *cellular* system such as ours concerns the configuration of our FPGA.

It should be clear from the description of the cells and of the MUXTREE elements that a network of cells, however limited, would nevertheless require a large number of such elements. In fact, the number of FPGA elements could be expected to rapidly reach proportions which would render a traditional configuration somewhat awkward, particularly where the generation of the configuration stream is concerned.

It would therefore be a not inconsiderable advantage if we could exploit the fact that our cells are identical, and that the configuration of the network consists in fact of the repetition of the same pattern throughout the array. Of course, this task is complicated by the fact that the size of the individual cells cannot be determined *a priori* without imposing unacceptable limitations to the versatility of the system. To address this question, we therefore needed to develop a mechanism which would allow us to "colonize" our FPGA, that is, to automatically fill the available array of MUXTREE elements with a pattern of our choice, a process which bears close resemblance to the biological concept of *self-reproduction*.

To develop such a mechanism, we selected an approach based on *cellular automata*, which had been used in the past to study the phenomenon of self-reproduction[2] [2, 4, 17]. After experimenting with "pure" cellular automata [15], we settled for a hybrid approach, in which an automaton is used to define the area to be occupied by the cell, and more traditional methods to actually charge the configuration.

The first step in configuring our FPGA is therefore to divide the array into identical squares (our experience with cellular automata revealed that the introduction

[2] Again, a certain confusion is generated by the improper (at least, in our context) use of the word "cellular", which we will try to avoid by replacing the word "cell" by the word "element", as we did when discussing FPGAs.

of rectangular structures causes an increase in complexity which, for the moment, we found excessive). We implemented this feature by adding a very simple automaton to our array (Fig. 7), which is configured by a stream of information entering the circuit from the lower left-hand corner. This stream, which consists of a simple repetition of an identical pattern, will cause the elements of the automaton to form a set of squares which will eventually completely fill the available space. The state of the elements of the automaton will then be used by the MUXTREE elements to direct the configuration stream (Fig. 8).

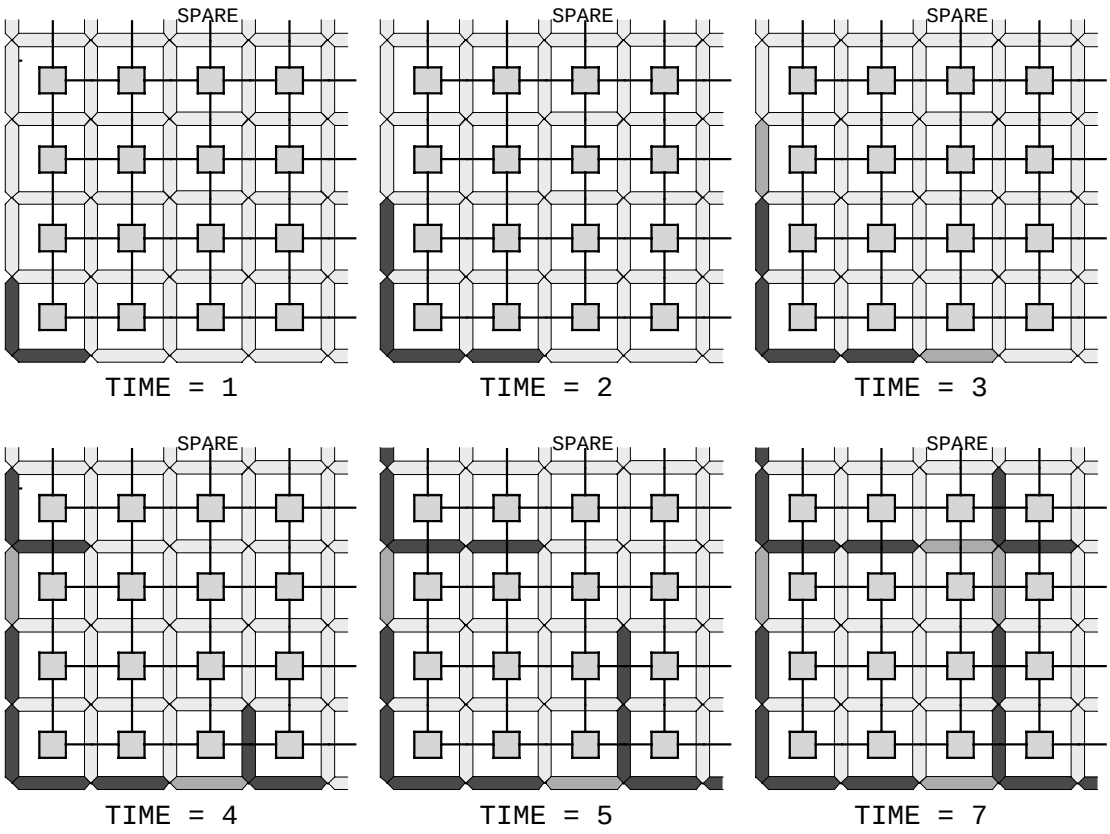

Fig. 7: Colonization of a MUXTREE array though the use of a cellular automaton.

A very interesting “bonus” to this approach lies in the possibility of programming the frequency of the spare columns. As shown in Fig. 7, the spare columns are defined by special states of the automaton, shown in paler gray in the figure, and are thus defined by the configuration stream, rather than being hardwired in the circuit. This approach lends a great flexibility to the system, since the frequency of the spare columns, and therefore the degree of robustness of the system, can be programmed into the configuration stream. The user is therefore free to trade space (i.e. number of usable cells) for robustness (i.e. number of spare columns) to fit the requirements of a particular application.

Once the squares are in place, in fact, it is relatively simple to actually configure the FPGA. The configuration stream will fill the pre-defined squares by weaving a path through the MUXTREE elements and configuring each register in turn, “skipping” the spare columns, which thus remain unconfigured until a fault occurs (Fig. 8).

With a relatively small amount of additional logic (the automaton itself is very simple), we can thus "colonize" the entire available surface of MUXTREE elements. In fact, we do not actually need to know the size of the FPGA if not to determine when the automaton has finished its task and the configuration of the FPGAs can begin.

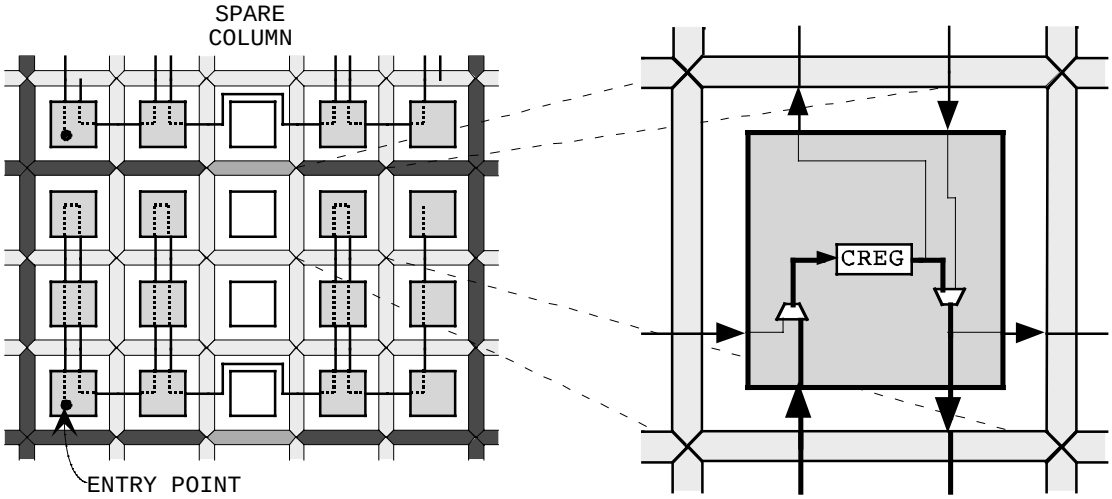

Fig. 8: Configuration path for a cell in a colonized MUXTREE array.

6. CONCLUSION

Biology and electronics are very different domains. The mechanisms of one cannot be easily applied to the other, even when they could prove very useful for certain applications [3]. However, with a careful analysis, it is sometimes possible to bridge the gap between the two domains, as we have tried to do with our system.

We developed an FPGA capable of self-repair and self-reproduction, based on a two-level approach where a set of small processors, loosely comparable to biological cells, is implemented on an FPGA of our design. This system is capable of self-repair at both levels, thus providing a kind of robustness which is not quite equivalent to that of biological systems, but does exploit some of its mechanisms.

Of course, from the point of view of standard circuit design, our system is far from optimal with respect to those parameters which are normally used to judge an implementation, namely speed and surface. However, it does provide us with the potential for a very powerful and very robust parallel system. Moreover, the experience we gained through this approach might well be applicable to more conventional circuit design methods. In particular, the self-test and self-repair system we developed for our MUXTREE circuits could, without a lot of effort, be adapted to other FPGAs, and the amount of logic required for self-repair, which is considerable in rapport to the size of a MUXTREE element, might become more acceptable if the mechanism is applied to more complex elements.

REFERENCES

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

[2] E. F. Codd, Cellular Automata (Academic Press, New York, 1968).

[3] R. A. Freitas and W. P. Gilbreath, "Advanced Automation for Space Missions", NASA Report CP-2255, 1982.

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

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

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

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

[8] R. Negrini, M. G. Sami, R, Stefanelli, Fault Tolerance Through Reconfiguration in VLSI and WSI Arrays (The MIT Press, Cambridge, Massachusetts, 1989).

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

[10] C. Piguet, "Tolérance aux pannes II", Technical Report, CSEM, Neuchâtel, September 1993.

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

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

[13] A. Stauffer, D. Mange, E. Sanchez, G. Tempesti, S. Durand, P. Marchal, C. Piguet, "Embryonics: Towards New Design Methodologies for Circuits with Biological-like Properties", in: *Proc. International Workshop on Logic and Architecture Synthesis*, Grenoble, December 1995, pp. 299-306.

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

[15] G. Tempesti, "A New Self-Reproducing Cellular Automaton Capable of Construction and Computation", in: *Advances in Artificial Life, 3rd European Conference on Artificial Life*, Granada, June 1995, Lecture Notes in Artificial Intelligence, Vol. 929 (Springer Verlag, Berlin, 1995) 555-563.

[16] N. Tsuda, T. Satoh, “Hierarchical Redundancy for a Linear-Array Switching Chip”, in: *Proc. 2nd IFIP Workshop on WSI*, Brunel, Sept. 1987.

[17] J. von Neumann, The Theory of Self-Reproducing Automata, A. W. Burks, ed. (University of Illinois Press, Urbana, 1966).

[18] M. Wang, M. Cutler, S.Y.H. Su, “On-Line Error Detection and Reconfiguration with Two-Level Redundancy”, in*: Proc. COMPEURO 87*, Hamburg, 1987, 703-706.