FPGA with selfrepair capabilities

Serge Durand
durand@lslsun.epfl.ch

Logic Systems Laboratory
Swiss Federal Institute of Technology
CH–1015 Lausanne, Switzerland

Christian Piguet

Centre suisse
d'électronique et de microélectronique SA
CH–2007 Neuchâtel, Switzerland

Abstract

In this paper, we present a new FPGA based on a multiplexer cell with selfdiagnosis and selfrepair capabilities. It is shown that the regular structure of FPGAs is particularly well suited for reconfigurable circuits. Error detection and reconfiguration mechanisms using spare cells are also presented.

1 Introduction

Redundancy techniques have been extensively used in the past decades. They were first intended to provide with highly reliable systems in areas where failure could not be tolerated. More recently redundancy has been recognized as a means to increase yield in large VLSI products, since yield exponentially decreases with chip size. Replicating some functional blocks in a chip and then selecting the good ones allows to get around silicon defects.

Regular architectures such as memories [1] and systolic processing arrays [2] are best suited for these techniques since their building blocks are identical. Therefore a small number of spare elements can be used to replace any defective element.

When it comes to random logic, these techniques are less efficient. As regularity in such structure does not exist, every block in the circuit has to be duplicated, thus leading to a large silicon overhead for guaranteeing at most one replacement for each element.

An FPGA permits to implement any logic function in a regular structure. Therefore we believe that FPGAs are a promising way to provide fault-tolerance for random logic designs.

Several reconfiguration techniques have been described in the literature and/or are used in today chip manufacturing. They can roughly be grouped in two main categories :

- reconfiguration using test and reconfiguration apparatus external to the chip
- reconfiguration using on-chip selftest and reconfiguration logic

Most of the systems described in literature or used in manufacturing processes are of the first type. They involve use of fuses either laser programmable [3], [4], [5] or electrically programmable [6]. Reference [7] shows that interconnections can even be established by a further masking stage after individual cells of the chip have been tested. Any of these methods require that all defects can be identified during the testing process. With circuits growing in complexity this is not always possible.

Circuits in the second category have the capability to reconfigure by themselves as long as sufficient spare material is available, not only during manufacturing process but also while they are operating within a system.

The FPGA presented in this paper belongs to the second category and is thus able to detect a failure during run-time and reconfigure itself dynamically.

2 Reconfiguration requirements

Reconfiguring a chip can be decomposed into the following steps:

1. detect an error;
2. locate the faulty cell;
3. choose a replacement cell for the defective one;
4. disconnect the faulty cell and connect the selected one instead;
5. update the chip configuration program;

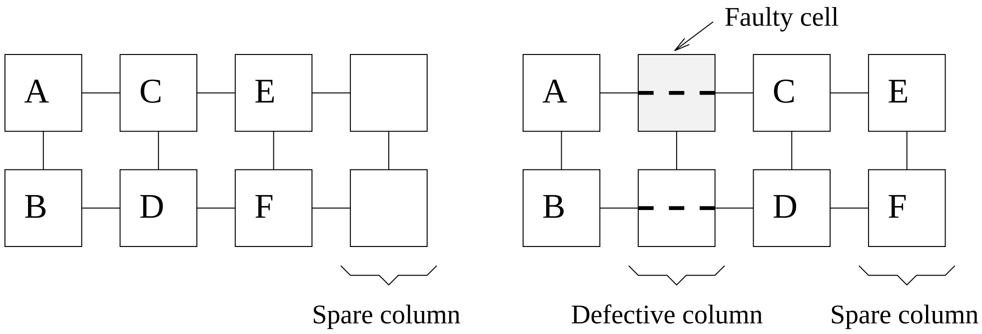

Figure 1: Bypassing a defective column

Since the reconfiguration must be accomplished by the chip itself, each of these steps must be as simple as possible. In particular the *choose and replace* algorithm will have a great influence on the chip architecture.

The philosophy of most of today's FPGA (Xilinx, Actel, ... [11]) is to have independant logic and routing resources. In this way any cell can be connected to any other by means of general purpose interconnect lines. Finding a path from one cell to another is a very complex task which demands important computing resources, and is even not always possible since these lines are limited in number. This kind of architecture is ill suited for our purpose.

Another architecture in vogue is an array in which only neighboring cells are connected together (Concurrent logic, Algotronix). Each cell has its own set of links and routing is part of the cell configuration. Figure 1 shows how connectivity can be preserved in such an architecture by simply bypassing a column containing a faulty cell.

Other strategies such as reconfiguring the array on a cell basis [7] or selecting between rows and columns in order to minimize the number of cells used [8] are more space efficient but lead to much more complex algorithms that are not suitable for silicon implementation.

Reconfiguring the chip after a faulty cell has been detected is accomplished by shifting configurations data of the cells located at the right of the faulty cell to the right. However care must be taken since defects are randomly spread across the chip and thus can destroy configuration memory as well as functional logic. It will be shown later how data can be secured.

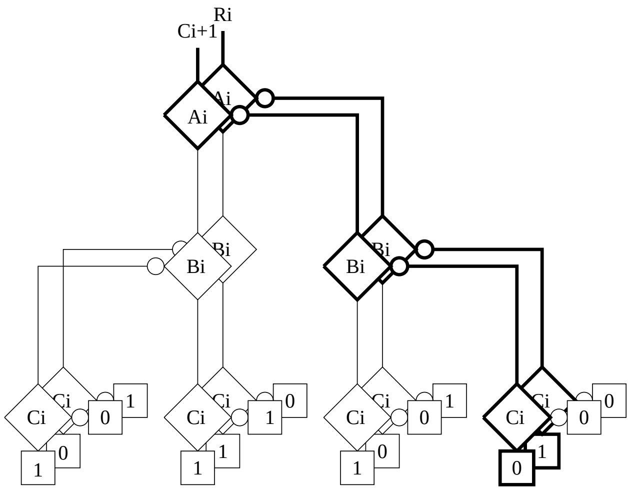

Figure 2: BDT of the 1 bit adder

3 Routing strategy

The success of an FPGA strongly depends on the associated placer and router performances. Our cell was designed in order to implement binary decision trees [10] [12] (BDT) for which good simplification algorithms exist. This methodology is well suited for our purpose since only neighboring test cells are connected together. The placement and routing is directly determined by the topology of the associated tree.

The example below shows how a single bit full adder slice can be implemented using BDTs.

The equations of the adder are:

$$R_i = A_i \oplus B_i \oplus C_i$$

$$C_{i+1} = A_i B_i + A_i C_i + B_i C_i$$

where A_i and B_i are the inputs, C_i the carry in, R_i the result and C_{i+1} the carry out bit.

These equations may be represented by two binary trees as shown in Figure 2. The leaves of the trees

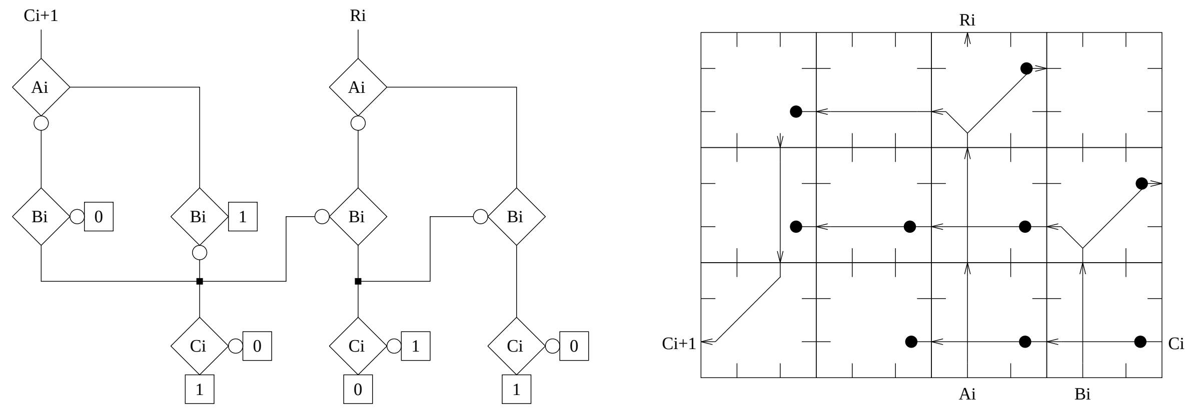

Figure 3: Reduced BDT of the one bit adder and second level interconnections

represent the value taken by the function when different input values are applied. Each of the diamonds represent a test on the variable represented inside the diamond. When the variable has the value '0' then the circled input is propagated toward the root, otherwise the other input is selected. Squares with a value inside represent constants. In Figure 2, bold lines show the selected path when inputs are set to $A_i = 0, B_i = 0, C_i = 1$. One can see that this combination of inputs produces the outputs $R_i = 1$ and $C_{i+1} = 0$.

The trees obtained in this way are said to be *canonical* since they possess a distinct branch for each distinct input set. They can be simplified by grouping identical subtrees as shown in the left part of Figure 3.

4 Basic cell architecture

In this section, the architecture of the basic cell is presented, without any reconfiguration and error detection logic. Reconfiguration and error detection will be introduced in sections 5 and 6.

The cell implements a test operator in the binary decision tree. Each of these tests is implemented using a multiplexer. As can be seen in Figure 2, inputs of a test are either constants (leaves of the tree) or outputs of other test operators (branches). In such a tree only neighboring connections are allowed. The inputs of a test are taken from the output of the south (*Sin*), south-east (*Ein*) or south-west (*Win*) test-operator.

In some cases one would like to connect the output of a tree to one input of another tree, even if they are not close to one another. For this purpose a second level interconnection scheme is provided. It features two directional busses in each direction and connects

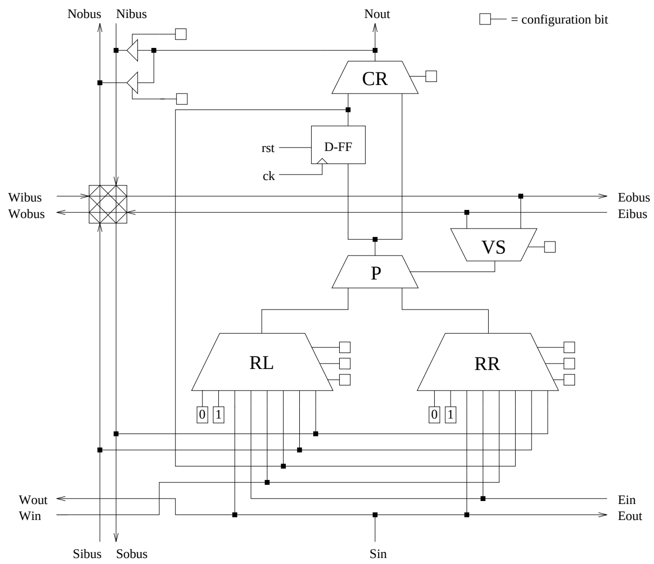

Figure 4: Architecture of the Muxtree cell

a cell to its four direct neighbors.

The test is performed by the main multiplexer labelled P in Figure 4 while two multiplexers labelled RR and RL select the source of the inputs. In addition to the connections mentioned above, the routing multiplexers may also take their inputs from the south busses *Sibus* and *Sobus*. The result of a cell is output to the north. In addition it can be fed to any of the north busses *Nibus* and *Nobus*. Each cell contains a D flip-flop which may be used for sequential circuits.

A connection box in each cell allows connections from any input bus to any output bus. Busses are also used to route variables through the chip. As variables are taken from the east busses, one should try to optimize bus usage by aligning cells using the same variable on rows.

Since trees are likely to contain equivalent subtrees, they should be simplified in order to decrease the number of cells used. The left part of Figure 3 shows the simplified trees for the adder example. The right part shows how variables are distributed among the cells. Black spots show where variables are taken from. Inputs and outputs of the adder are placed in such a way that a n-bit adder can be formed by joining several parts horizontally. Free busses may be used to route signals between trees.

5 Error detection

Faults at the physical level may result in *errors* at the output of the circuit. If the result at the output of the circuit is not equal to what was expected, then a fault must be present in the circuit. Although one can locate the fault by merely observing the outputs when applying test-vectors to the inputs, we decided to include fault detection logic at the cell level. This permits to test the chip continuously and detect errors as soon as they occur.

To test whether the one bit output of a cell is correct, one must have a reference value to compare with. This second value is produced by duplication of part of the cell. If both values were to be evaluated in parallel, all the logic should be duplicated. Since routing muxes are already present in two identical samples, it is advantageous to evaluate values in sequence, and thus avoid to duplicate them. Figure 5 shows the modified architecture of the cell.

Computing a new value is done in two steps. First the `tst` signal is set to high. This is the normal operation for the routing multiplexers: mux RL feeds mux P1 with the '0' (false) input while mux RR provides it with the '1' (true) input. The output of the P1 mux is stored in flip-flop D1.

Signal `tst` is then set to low which reverses the role of the routing multiplexers by changing their selection inputs using the configuration selector muxes. Mux RL now feeds mux P2 with the signal that originally went through RR during phase one. The opposite occurs for mux RR. This inversion is canceled by mux P2 and the result is then stored in flip-flop D2. If the contents of flip-flop D1 and D2 are equal the next evaluation can be processed since no error occurred. This of course doesn't mean that the circuit is fault free, only that a faulty path has not been used.

On the other hand, if the content of the flip-flops are not equal, a fault *must* be present in the cell and

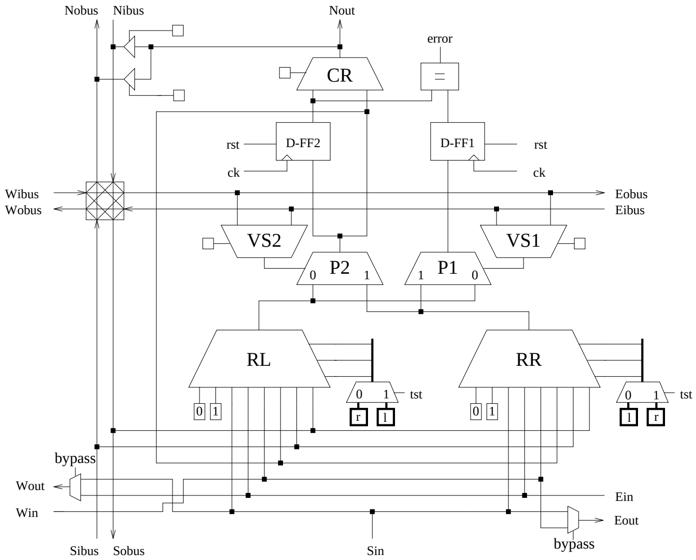

Figure 5: Error detection logic

the circuit must be reconfigured, thus the `error` signal is set to high.

6 Configuration logic

17 bits are needed to configure the cell : 10 for the muxes, 7 for the connection box. Implementing the configuration memory as a shift register is particularly efficient for reconfiguration as will be shown.

Since the chip must be able to reconfigure without accessing external elements, it must be ensured that even after a failure has occurred, possibly in the configuration memory, the entire configuration can be rebuilt. It is realized by providing each cell with two identical independent registers with parity checking. Each configuration word has its odd parity bit appended to the right side.

Sequential parity computation is achieved by a single T flip-flop, whereas parallel computation would need an XOR tree. Computing the parity is done by cycling the configuration word and feeding the input of the T flip-flop with the output of the configuration register.

The hypothesis that only one fault occurs at a time is usually made. Since each cell contains two configuration registers, it is certain that at least one of the registers contains the correct configuration bits at any time.

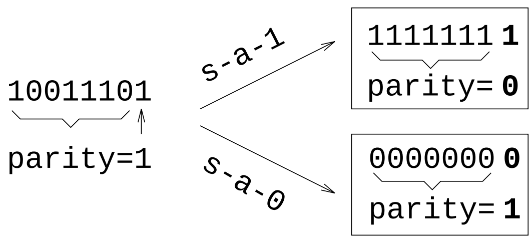

Figure 6: Stuck-at errors

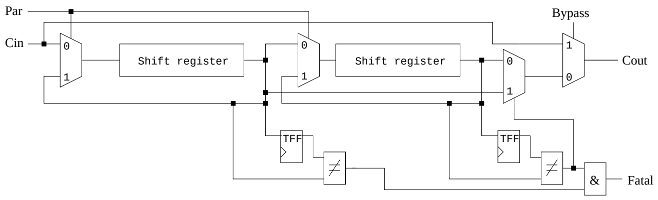

Figure 7: Configuration logic

7 Configuring the array

During configuration, care must be taken so that faults are not present in the configuration memory, since this would result in errors that could possibly not be detected. Moreover since the configuration memory is built as a shift register, a stuck-at fault in one register would corrupt all the bits flowing through. For this reason, each register must be checked as soon as it has been loaded and the whole column discarded if the content is erroneous. Discarding a column is achieved by asserting the `bypass` signal.

Since most failures at the transistor level result in stuck-at faults, all the bits encountering the same fault are likely to be altered in the same way, that is all turned into 1 or 0. When rotating through the register in presence of a fault, all the bits take on the same value. Thus, including an odd parity bit into each configuration word permits to detect such stuck-at faults, since an all-one word should have a parity bit set to zero and vice-versa (as long as the word contains an odd number of bits). See Figure 6.

8 Reconfiguration

In the proposed architecture, reconfiguration is achieved by shifting to the right the configuration of all the columns located on the right of the faulty column (Figure 1). The defective column is then simply bypassed.

Since the fault may be located in the configuration memory, it must first be determined which of the two

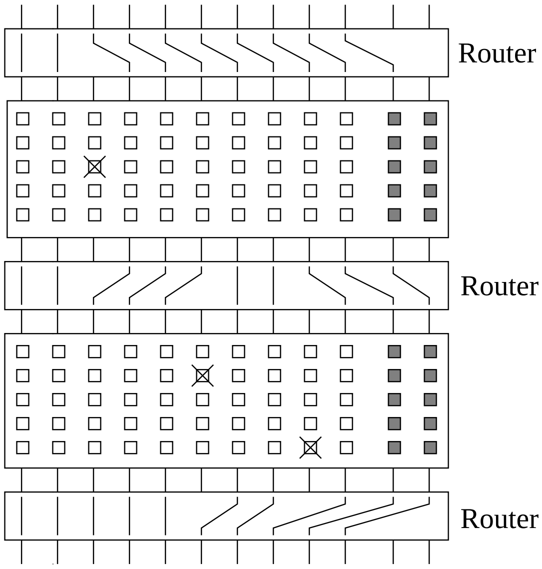

Figure 8: Cell array

words is correct (if not both). Again this is done by computing the odd parity of the configuration words and comparing with the parity bit stored in the configuration word.

If none of the configuration words is correct then the chip cannot reconfigure and the `fatal` signal is asserted.

9 Optimal array size

Since the array is reconfigured by replacing columns rather than rows, it is better to have short columns and long rows. In this way each reconfiguration will discard a minimum number of cells. However, a minimum column height is needed in order to build large trees without cutting them into too many sub-trees. To meet both requirements, each column can be split into several sub-columns that can be independently reconfigured. Given an area with a given number of defects, half this area will contain half the defects. This implies that having n independent functional arrays increases the tolerance of the chip by a factor of n, or that the number of spare columns can be reduced by a factor n for a given density of defects.

Inputs and outputs of the chip are fixed in contrast to the cell array which is reconfigurable. In the array a given signal may come out at different places according to the number of reconfiguration that were performed, on the other hand, this signal is assigned to a pin of the circuit and cannot be moved. Therefore a router must be inserted between the array and the pads. The same problem occurs between independent sub-arrays and the same precaution must be taken. Figure 8 shows an overview of the array.

10 Reconfiguration overhead

Total overhead for the chip is given by :

$$overhead = \frac{2 \cdot arraysize + sparesize}{arraysize}$$

Realistic defect densities lie between 1 and 5 defects/mm^2. For a 400 cells chip, which would lead to 80 mm^2 in a 0.8μ technology, up to 4 defects should be expected, and thus, 4 spare columns should be provided to balance these early defects. More spare columns may be provided for further reconfigurations. Four spare columns lead to an overhead of about 110% for a square array. Two more columns lead to 115% overhead. An overhead of 100% can be approached by a circuit having more columns than lines or by splitting the array into several sub-arrays as seen before.

11 Conclusion

The area overhead may seem prohibitive, however, it provides our FPGAs with a unique feature : selfrepair capability.

With the continuous improvements of technology, transistor count will be less and less concerning in chip design. Therefore, it may be expected that in a near future, doubling the chip size just for the sake of repairability will no more be seen as an heresy.

References

[1] S. E. Shuster, *"Multiple word/bit line redundancy for semiconductor memories"*, IEEE J. Solid-State Circuits, vol. SC-13, p.693, 1978.

[2] T. Leighton, C. E. Leiserson, *"Wafer scale integration of systolic arrays"*, IEEE Trans. Comput., vol C-34, p.448, 1985.

[3] J. I. Raffel *et al.*, *"A demonstration of very large area integration using laser restructuring"*, Proc. IEEE Int. Symp. Circuits and Systems, p.781, 1983.

[4] J. I. Raffel *et al.*, *"A wafer-scale digital integrator using restructurable VLSI"*, IEEE J. Solid-State Circuits, vol. sc-20, p.399, 1985.

[5] R. P. Cenker *et al.*, *"A fault tolerant 64K dynamic random access memory"*, IEEE Trans. Electron Devices, vol. ed-26, p.853, 1979.

[6] K. Kokkonen *et al.*, *"Redundancy techniques for fast static RAMs"*, ISSCC, p. 80, 1981.

[7] K. Yamashita *et al.*, *"A wafer-scale 170000 gate FFT processor with built-in self test"*, IEEE J. Solid State Circuits, vol. sc-23, p.336, 1988.

[8] J. A. Abraham, *et al.*, *"Fault tolerant techniques for systolic arrays"*, IEEE Computer, vol. 20, no. 7, p.65, 1987.

[9] A. Stauffer, P. Marchal, *"Binary decision tree oriented FPGAs"*, FPGA '94, Proc. ACM International Workshop on Field-Programmable Gate Arrays, submitted

[10] D. Mange, *"Microprogrammed systems"*, Chapman & Hall, London, 1992, ISBN 0-412-40800-7

[11] J. Rose *et al.*, *"Field programmable gate arrays"*, Kluwer Academic Publishers, 1992, ISBN 0-7923-9248-5

[12] D. Mange, *"Life in silico"*, ECCTD '93, Proc. Circuit Theory and Design, Davos, 1993