Presented at the
International Conference on Computer Design
Port Chester, New York, November 1983.

SELF - CHECKING VLSI BUILDING BLOCKS FOR FAULT - TOLERANT MULTICOMPUTERS

Yuval Tamir and Carlo H. Séquin

Computer Science Division
University of California, Berkeley, CA 94720

Abstract

The use of self-checking nodes and links for implementing fault-tolerant VLSI multicomputers is proposed. The system is composed of a large number of VLSI computers interconnected by high-speed dedicated links. Hardware that performs error detection is combined with system-level protocols that handle error recovery and fault treatment.

The self-checking nodes notify the rest of the system when their output is erroneous. In order to achieve high fault coverage, error detection is accomplished by duplication and matching. The critical circuit in this scheme is a comparator which must not be susceptible to faults that can remain undetected and later mask the failure of the functional modules. With both NMOS and CMOS technologies it is possible to implement a *self-testing* comparator that will produce an error indication if it incurs any single physical defect.

Introduction

High reliability and high performance are primary goals of most computer systems. There are fundamental limits to the increases in reliability and performance that are achievable by improvements in technology alone. The limits on performance can be overcome by exploiting parallelism while the limits on reliability can be overcome by using *fault tolerance* techniques.

Parallelism can be exploited by a system that consists of a large number of *computation nodes*, each able to execute a subtask of the problem being solved. A possible architecture for such a system, which is compatible with the constraints of VLSI, is to interconnect these computation nodes by high-speed dedicated links and *communication nodes* that provide hardware support for communication functions such as message routing. Each computation node is connected to one of the communication nodes. A communication node has several *ports* through which it is connected to computation nodes and other communication nodes.[9] We call such a system a *multicomputer*. The nodes and links are components (building blocks) that can be used to construct multicomputers with a wide range of sizes and topologies.

System *failure* occurs when the multicomputer doesn't perform according to its specifications at its interface with the "outside world." System failure is often the result of a failure of one of its components. Fault tolerance techniques attempt to prevent component failure from leading to system failure.[1] A multicomputer is especially well suited for reliability enhancement using fault tolerance techniques since it is partitioned into independent and "intelligent" components (the computation and communication nodes). Fault-free components can adapt to changes in faulty components and continue their operation in a way that leads to correct system output despite the fault.

A brief overview of techniques for implementing fault tolerance in a multicomputer is presented and the considerations that lead to the choice of an approach based on self-checking components are discussed. *Duplication and matching* is shown to be an effective practical technique for implementing nodes that are self-checking with respect to any likely fault.

Implementation of Highly Reliable Multicomputers

The reliability of any system can be enhanced by increasing the reliability of its components through *fault prevention*[1] techniques such as specialized design methodologies, stringent quality control, and extensive validation and testing. These techniques typically result in more complex designs, greater cost, and lower performance. Furthermore, the effectiveness of these techniques is limited by our inability to exhaustively test complex VLSI chips.[8]

Alternatively, the reliability of the components can be increased by employing *fault tolerance* techniques. These techniques attempt to ensure that each component will continue to perform according to its specifications despite faults. Unfortunately, no component can tolerate an unbounded number of faults. The contamination of the system by incorrect output from a faulty component can be prevented only if, at some stage, other system components find out about the failure of the component and physically or logically isolate it from the rest of the system.

At the system level, software (protocols) can be used to detect and recover from the failure of components. For example, identical tasks may be assigned to three nodes and a "majority vote" taken on the results. One of the problems with this approach is that if the results conflict, it may be very costly or impossible to locate the cause of the discrepancy. Additional problems are the high overhead in computation resources and communication bandwidth and difficulties in effectively handling transient faults.

If a node fails due to a transient fault, it should be reset to a "sane state" and remain active rather than be removed from the system. If neighboring nodes are responsible for detecting such a failure, they must be given the authority to initiate the reset. However, this authority also allows a *failed* node to reset operational neighbors. In order to prevent this situation, each node must be responsible for its own reset. Hence, the node should include a mechanism to detect its own *erroneous states*[1] and initiate the reset.

Some of the deficiencies with the aforementioned techniques can be overcome by implementing fault tolerance in a VLSI multicomputer using hardware *error*

CH1935-6/83/0000/0561$01.00 © 1983 IEEE

detection in conjunction with system level protocols which perform *error recovery* and *fault treatment*.[1] Errors caused by faults in the communication links are detected through the use of error-detecting codes. All nodes are self-checking and signal to the rest of the system when their output is incorrect so that it will not be accepted as correct. In addition, failed nodes attempt to reset themselves and reestablish a sane state. The immediate neighbors are informed whenever a node fails. If the node doesn't reset itself or fails too often, the neighbors can logically remove it from the system by refusing to communicate with it. The diagnostic status information is distributed throughout the system so that, eventually, no fault-free node will attempt to use the faulty component.

Self-Checking Nodes

For all likely faults, a self-checking component must either produce the "correct" outputs or somehow indicate that the outputs are incorrect. A component that satisfies this requirement is said to be *fault secure*.[10] If the component is not guaranteed to produce an error indication immediately following the first fault, it is possible for several faults to exist in the component simultaneously without any indication to the rest of the system. Even if the component is fault secure with respect to any single fault, several faults together may lead to the failure of the self-check mechanism and, eventually, to incorrect outputs from the component being accepted as correct by the rest of the system. In order to prevent this situation, the component must be *self-testing*.[10] In the presence of one or more faults, a self-testing component produces an error indication before additional faults can occur and lead to the failure of the self-check mechanism. Components which are fault-secure and self-testing are said to be *totally self-checking*[10] (*TSC*).

Error detecting/correcting codes can be used to implement *TSC* nodes. Redundant information is carried by busses, memories, and registers in order to detect (and possibly correct) errors.[10] Unfortunately, different coding schemes must be used for different parts of the node. The resulting increase in the complexity of the design and of design verification and testing may lead to a circuit in which failure modes that are more difficult to predict and "tolerate" are more likely to occur.

An alternative is to construct the *TSC* computation or communication node using two identical, *independent* modules, each performing the function of the node. Inputs from neighbor nodes are fed to both modules. If the modules operate synchronously, their outputs should always be identical. Except for the nearly-impossible case where both modules produce identical *incorrect* output, an error can be detected by a comparator which is part of the node. The output of the comparator is connected to neighboring nodes through dedicated wires. The output from one of the two modules is the "functional" output from the node (Fig. 1). A "no-match" signal from the comparator is used locally as a reset signal and is also sent to all neighbors as a failure indicator. Similar failure indicators from the neighbors cause an interrupt and invoke system-level routines that handle the node failure.

Implementing the *TSC* property in a component using duplication and matching may appear wasteful since it more than doubles the required hardware. However, this scheme becomes more attractive when issues such as design complexity, fault coverage,

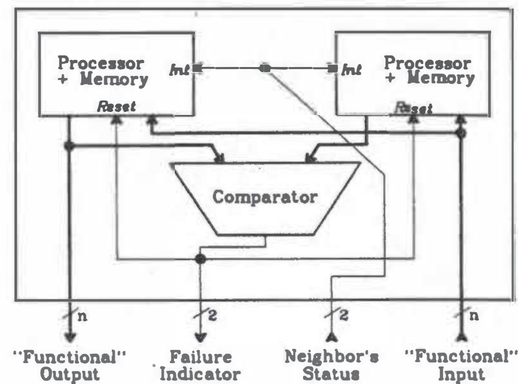

Fig. 1: *A Self-Checking Computation Node*

reliability prediction, and the ability to recover from transient faults are taken into account. Traditional fault models are not valid for VLSI.[5,12] As a result, low-cost error detection schemes, which are based on these models, may no longer be adequate. With duplication and matching, errors are detected as long as the comparator remains functional and the two modules produce different outputs the first time one or both of them fail. Since a faulty comparator can mask faulty functional modules, faults in the comparator must not go undetected, *i.e.*, the comparator must be self-testing. Thus a detailed analysis of the effects of all likely faults on the comparator is required.

Physical Defects and Logical Faults in VLSI

The design of self-checking circuits requires an understanding of the physical defects which commonly occur in VLSI and of the resulting logical faults. In the past the stuck-at fault model has been widely used to model, at the logical level, the effects of physical defects in circuits. This model does not cover many of the possible defects in VLSI.[3,5,12] The fabrication flaws and physical processes that can cause malfunction of NMOS and CMOS VLSI circuits are summarized in this section.

VLSI chip failures may be caused by design or fabrication flaws, may be due entirely to environmental factors, or are the end result of a degenerative process due to operational and environmental stresses but partially attributable to design or manufacturing defects.[4,10] Fabrication defects in MOS chips consist mainly of shorts and opens in each interconnection level, shorts between different levels, and large imperfections such as scratches across the chip.[5] Other fabrication defects include incorrect dosage of ion implants, contact windows that fail to open, misplaced or defective bonds, and penetration of the package by contaminants.[4] During the operation of the chip, faults may be caused by electromigration, corrosion, electrical breakdown of oxide, cracks due to thermal expansion, power supply fluctuation, and ionizing or electromagnetic radiation.[4]

At the logical level, most of the faults can be represented in a circuit model that consists of a network of switches, loads (for NMOS), and interconnection lines which directly correspond to the transistors and interconnections in the actual circuit.[5] Most of the physical defects, such as opens and shorts, can be represented in this model in an obvious way.[3] A "switch" may be permanently on or permanently off, corresponding to a gate input stuck-at-1 or 0, respectively. Shorted NMOS loads (pullups) are equivalent to an output line s-a-1. Disconnected gate inputs are usually equivalent to s-a-0 or s-a-1 faults.

Some physical defects have a more complex effect on the circuit. In NMOS, incorrect dosage of ion implants may cause a threshold shift in a load transistor. This can result in an output voltage that lies between the voltages assigned to logic 0 and logic 1. If the fanout from the gate is greater than one, some of the gates connected to its output may "interpret" it as logic 1 while others will interpret it as logic 0. If, at some point in time (clock cycle), the line is supposed to be a logic 1 but is interpreted by some of the gates as logic 0, we call it a *weak 1* fault. Conversely, if the line is supposed to be a logic 0 but is interpreted by some of the gates as logic 1, we call it a *weak 0* fault. A single physical defect, resulting in a single weak 0 or weak 1 fault, has the same effect as multiple s-a-1 or s-a-0 faults, respectively.

In CMOS, a transistor which is permanently off or a break in a line can result in a high impedance state where the output of a combinational logic gate is dependent on the previous output rather than the current input.[12] Such a fault (called a *stuck-open* fault) may escape detection even if all possible input vectors are used to test the circuit.[12]

Implementation of Self-Testing Comparators in VLSI

The duplication and matching scheme relies entirely on a self-testing comparator to detect faults in the functional modules. Implementing such a comparator requires knowledge of how different faults will affect the circuit. Fortunately, a comparator is a simple circuit that can be implemented with a regular structure and is therefore amenable to thorough analysis. Hence, we can have confidence in our ability to predict the likely physical defects, develop a valid fault model, and prove that the implementation we propose is indeed self-testing.

We assume that physical defects in the node occur one at a time. A fault that is the result of a single physical defect is called a *single fault*. It is assumed that there is a negligible probability that the time interval between the occurrence of successive single defects in the comparator or between a single defect in the comparator and an arbitrary collection of defects in the functional modules, is less then some value T. In order to ensure that faults in the comparator will not mask future faults in the functional units, during normal operation, the comparator must "test itself" for any single fault in less than time T.

Single Stuck-At Faults

As a first step to constructing a comparator which is self-testing with respect to *any* single fault, we will discuss the implementation of a comparator which is self-testing with respect to any single *stuck-at* fault.

In this context "two-rail" codes prove useful. They consist of all words (bit vectors) such that a specified half of the word is the complement of the other half. If the output of one of the modules in a self-checking node is complemented, a two-rail code checker can serve as a "comparator" that checks the validity of the output. Such a checker, which is self-testing with respect to any single stuck-at fault, can be implemented as a two level NOR-NOR PLA (Fig. 2).[13,2] The output from the checker is a two-bit two-rail code that is 01 or 10 (*code output*) if the input is a two-rail code word (*code input*), and 00 or 11 (*noncode output*) otherwise (*noncode input*). It can be shown that if any single stuck-at fault exists in the checker, there is an input two-rail code word that results in a 00 or 11 output, thereby "detecting" the fault.[13]

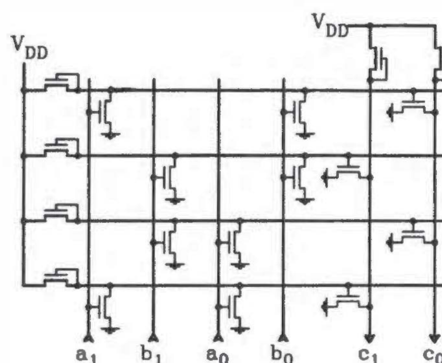

Fig. 2: *An NMOS Self-Testing Two-Rail Code Checker*

The requirement that the checker must be self-testing with respect to any single stuck-at fault poses severe constraints on its implementation. It can be shown that *any* two level AND-OR (or NOR-NOR) implementation for an input of $2n$ bits (n bits from each module) must use 2^n product terms, one for each code input.[11] If the output from each module is, say, 16 bits, this implementation is impractical since it requires $2^{16} = 65536$ product terms. Furthermore, all possible (2^n) code words must appear at the checker's inputs for it to perform a complete self-test.

Several small self-testing two-rail code checkers can be used as "cells" for constructing a self-testing checker for a wide input word (Fig. 3).[10,6] While the self-testing property is preserved, the number of input patterns required for a complete self-test is dependent only on the size of the largest "cell."[6]

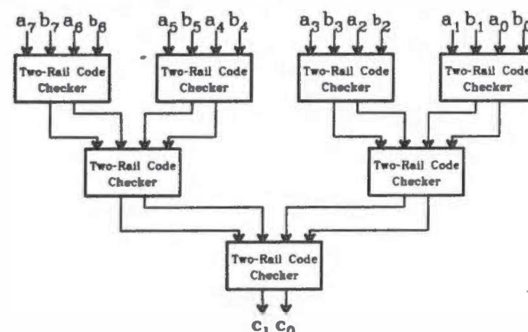

Fig. 3: *A Self-Testing Two-Rail Code Checker Tree*

Other Single Faults

The faults that commonly occur in a MOS PLA are stuck-at faults, shorts between adjacent lines, breaks in lines, and contact faults which include missing or extra devices at crosspoints.[13,7] In addition, weak 0/1 faults can occur on the input or product term lines. Fortunately, it turns out that a straightforward NOR-NOR PLA implementation of the checker discussed above is self-testing with respect to any one of the aforementioned single faults. The rest of this section contains an informal "proof" of this claim; a more formal proof will be presented elsewhere.[11] Faults in the input lines, product term lines, output lines, AND array crosspoints, and OR array crosspoints, are considered separately.

Any single stuck-at fault or short in the input lines will cause one or more 0's to change to 1's *or* one or more 1's to change to 0's (but not both) for some code input. It can be shown that such an error (called a

unidirectional error[7]) on the input lines results in noncode output.[13] A break in the input line outside the AND array is equivalent to the line stuck-at-0 or stuck-at-1. A break in the middle of the AND array affects only some product terms. For an affected product term, if the break is equivalent to a stuck-at-1, the one code input that is supposed to select this product term won't, and a noncode output will result. If the break is equivalent to a stuck-at-0, there exists a code input that results in a noncode output since it selects two product term lines each of which is connected to a different output line.[11]

An extra device in the AND array is equivalent to the corresponding product term stuck-at-0. The code input that is supposed to select that product term results in a noncode output. If there is a missing device in the AND array, there exists a code input that produces a noncode output since it selects two product term lines, each of which is connected to a different output line.[11]

An extra device in the OR array means that one of the product terms is connected to both outputs. A missing device in the OR array is equivalent to the corresponding product term stuck-at-0. In either case, the code input that selects the relevant product term will result in a noncode output.

If the output lines are shorted, their values are equal and that is a noncode output. If one of the lines has a stuck-at fault, there exists a code input that causes the other line to have the same value so the output is noncode. For some code input, a break in one of the output lines is equivalent to a stuck-at-1 or stuck-at-0 fault on that line.

A stuck-at-0 fault on a product term line will result in a noncode output if the input is the code word that is supposed to select that product term line. A stuck-at-1 fault on a product term line will result in a noncode output to any input that selects a product term line that is connected to the other output line. A break in a product term line is equivalent to a stuck-at fault on that line since each product term line is connected to only one output line. A short between two product term lines will result in a noncode output if the input selects either one of these lines.[11]

Product term lines are not susceptible to weak 0/1 faults since each product term line is connected to only one output line (fanout of one) so that a weak 0/1 fault is equivalent to a *single* stuck-at fault. Input lines have a fanout greater than one and are thus susceptible to weak 0/1 faults. A weak 1 fault on an input line is equivalent to one or more missing devices in the AND array. Each product term that is connected to a "missing device" will be selected by an input code word that also selects a product term line that is connected to the *other* output line.[11] Thus, a noncode output will result. A weak 0 fault on an input line is equivalent to one or more product term lines which are stuck-at-0. Any code input that is supposed to select one of these product terms will result in a noncode output.

In CMOS chips PLAs are usually implemented in dynamic "pseudo NMOS."[12] All product term and output lines are precharged during every clock cycle before being selectively discharged according to the input. Therefore no state is preserved from one cycle to the next and the circuit is combinational despite any opens in the precharge or discharge paths.[11] Hence the PLA used in CMOS chips is only susceptible to the same faults as the traditional static PLA used in NMOS chips.

This analysis shows that for all single faults in our fault model, there exists a code input that results in a noncode output from the proposed two-rail code checker PLA. Thus, the checker is self-testing with respect to any likely single fault. Based on this result, it can be shown that the checker constructed as a tree of smaller self-testing checkers (Fig. 3) is also self-testing with respect to any likely single fault.[11]

Summary and Conclusions

We have presented an approach to increasing the reliability of future high-end systems beyond what is possible with technological solutions alone. The system consists of computation nodes and communication nodes, interconnected by high-speed dedicated links. These components are relied upon to detect errors while system level protocols are used for error recovery and reconfiguration.

The use of duplication and matching for implementing the self-checking nodes allows us to restrict a detailed analysis of the impact of all possible faults to the comparator, which is a relatively simple circuit. We have shown that the self-testing comparator, which is the backbone of our approach, can be implemented with NMOS and CMOS technologies.

Acknowledgements

We would like to thank Richard Fujimoto, Manolis Katevenis, and Robert Sherburne for reviewing a draft of this paper.

This research was supported by the State of California MICRO program and the Defense Advance Research Projects Agency (DoD), ARPA Order No. 3803, monitored by Naval Electronic System Command under Contract No. N00039-81-K-0251.

References

1. T. Anderson and P. A. Lee, "Fault Tolerance Terminology Proposals," *FTCS12*, pp. 29-33 (June 1982).
2. W. C. Carter and P. R. Schneider , "Design of Dynamically Checked Computers," *IFIPS Proc.*, pp. 878-883 (August 1968).
3. B. Courtois, "Failure Mechanisms, Fault Hypotheses and Analytical Testing of LSI-NMOS (HMOS) Circuits," pp. 341-350 in *VLSI 81*, ed. J. Gray, Academic Press (1981).
4. E. A. Doyle, "How Parts Fail," *IEEE Spectrum* 18(10) pp. 36-43 (October 1981).
5. J. Galiay, Y. Crouzet, and M. Vergniault, "Physical Versus Logical Fault Models MOS LSI Circuits: Impact on Their Testability," *IEEE-TC* C-29(6) pp. 527-531 (June 1980).
6. J. Khakbaz and E. J. McCluskey, "Concurrent Error Detection and Testing for Large PLA's," *IEEE JSSC* SC-17(2) pp. 386-394 (April 1982).
7. G. P. Mak, J. A. Abraham, and E. S. Davidson, "The Design of PLAs with Concurrent Error Detection," *FTCS12*, pp. 303-310 (June 1982).
8. R. A. Rasmussen, "Automated Testing of LSI," *Computer* 15(3) pp. 69-78 (March 1982).
9. C. H. Séquin and R. M. Fujimoto, "X-Tree and Y-Components," in *Proc. Advanced Course on VLSI Architecture*, Univ. of Bristol, England, ed. P.C. Treleaven, Prentice Hall (1982).
10. D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press (1982).
11. Y. Tamir, "Fault Tolerance for VLSI Multicomputers," Ph.D. Dissertation (in preparation).
12. R. L. Wadsack, "Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits," *BSTJ* 57(5) pp. 1449-1474 (May-June 1978).
13. S. L. Wang and A. Avizienis, "The Design of Totally Self Checking Circuits Using Programmable Logic Arrays," *FTCS9*, pp. 173-180 (June 1979).