Postgraduate Study Report DC-PSR-2004-07

Design of Self Checking Circuits Implemented in FPGA

Pavel Kubalík

Supervisor: *Ing. Hana Kubátová, CSc.*

January 2004

Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13
121 35 Prague 2
Czech Republic

email: xkubalik@fel.cvut.cz
WWW: cs.felk.cvut.cz/~xkubalik

This report was prepared as a part of the project

Design of Highly Reliable Control Systems Built on dynamically Reconfigurable FPGAs

This research has been supported by grant GA102/03/0672, MSM 212300014, GA102/04/0737, GA102/04/2137 and by foundation Nadání Josefa, Marie a Zdeňky Hlávkových

. . . .
Pavel Kubalík
postgraduate student

. . . .
Ing Hana Kubátová, CSc.
supervisor

Contents

1 Introduction **2**

2 Theoretical Background **4**
2.1 Self-Checking Design 4
2.2 Design of Functional Blocks 4
2.3 Fail-Safe Design 5
2.4 Error detection codes 6
2.4.1 Parity code 6
2.4.2 Dual-Rail code 7
2.4.3 M-out-of-n code 7
2.4.4 Berger code 7
2.4.5 Arithmetic codes 7
2.4.6 Hamming code 7
2.5 Design of Checkers 8
2.5.1 Code-Disjoint 8
2.5.2 Self-Testing 8
2.6 Perturbation Tolerant Memories 9

3 Related work **10**

4 Overview of our approach **13**
4.1 Error detecting codes 13
4.2 Fault model 14
4.3 Basic idea 15
4.4 Hamming code 15
4.5 Design scheduling 17
4.6 Software solution 17
4.7 Check bits generator 18

5 Experimental results **20**
5.1 Used benchmarks 20
5.1.1 Even Parity 20
5.1.2 Double even Parity 21
5.1.3 Hamming code (63,57) 21
5.1.4 Hamming code (255,247) 22
5.1.5 Partial conclusion 22

6 Conclusions and future work **23**

7 Dissertation thesis **27**

8 Publications of the author **28**

Design of Self Checking Circuits Implemented in FPGA

Pavel Kubalík
xkubalik@fel.cvut.cz
Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13
121 35 Prague 2
Czech Republic

Abstract

This report focuses on error detection in circuits implemented in FPGAs. We have used error detection codes (ED codes) to ensure self-checking property. The fault of given combinational circuit has to be detected and signalized at the time of its appearance and before further distribution of errors. Hence a safe operation of the designed system is guaranteed. The ability to detect an error during normal circuit operation is called concurrent error detection (CED). The check bits generator and the checker were added to the original combinational circuit ensuring the Totally Self-Checking property (TSC). In order to simplify testing only combinational circuits are used. Our work is based on previous work of others, who have used benchmarks that were described by tables. In some cases benchmarks with many inputs cannot be described by tables. The benchmarks, used in this work in order to compute a quality of the code, are described by equations. All of the experiments assume future implementation in XILINX FPGA circuits. Due to their further implementation in FPGAs the fault model considers the way the configuration data is stored in the configuration memory. This work is a part of a more complex methodology of a fault tolerant design based on FPGAs with dynamical reconfiguration of the faulty part of the designed circuit.

Keywords

on-line testing, self-checking circuits, fail-safe circuits, error detecting codes, FPGA

Chapter 1

Introduction

VLSI testing was dominated by the needs of achieving high quality manufacture testing with acceptable cost. With the rapidly increasing complexity of VLSI circuits this goal became increasingly difficult and had biased the effort of the test community on the direction of manufacturing testing [12].

However, important industrial applications require protection against field failures requiring on-line testing solution. These needs concerned at a first time specific products destined to safety critical applications and fault tolerant computing, which correspond to low volume production. At the same time the low number of these applications did not make attractive for CAD vendors the development of tools specific to the design of on-line testable ICs. The lack of such tools increases dramatically the effort for designing on-line testable ICs. The low-volume production of such applications often does not justify such a high development cost, since it will impact dramatically the per product unit cost. As a matter of fact, techniques using off-the-shelf components, such as duplication or triplication are more often adopted, since they represent a much lower development cost although the production cost is relatively high.

We can expect this situation to be changing. Various industrial sectors have rapidly increasing needs for on-line testing. Such sectors are for instance rail-way control, satellites, avionics, telecommunications, control of critical automotive functions, medical electronics, industrial control etc. Further, we can expect wider sectors of the electronics industry to be demanding for on-line testing solutions in order to ensure the welfare of the users of electronic products. Some of these applications concern high volume production and should support the standardization of such techniques, in the same way the increasing needs of VLSI testing have transformed DFT and BIST into standard design techniques, and have supported the development of specific tools today offered by most of CAD vendors.

Since silicon is "cheap", such tools should make very popular the design of on-line testable circuits. In addition to these trends, the high complexity of nowadays systems, require more efficient solutions. In fact complex multi-chip systems of yesterday are today single-chip components. As a matter of fact fault tolerant and fail-safe system designs of yesterday have to be integrated on chip level, appealing for on-line testing techniques for VLSI.

A large variety of on-line testing techniques for VLSI was developed in the past and are still enriched by new developments. They can respond efficiently to the needs expressed above, under the condition that available CAD tools simplify their implementation. Such

techniques are for instance: self-checking design, signature monitoring

The report is organized as follows. Section 2 introduces basic definitions and terminology. Section 3 summarizes the previous results. Section 4 defines the main framework for our solution. Section 5 discusses the results and Section 6 concludes with outlines for future work.

Chapter 2

Theoretical Background

2.1 Self-Checking Design

Concurrent checking verifies circuits during their normal operation. Because the output delivered by a circuit during its operations as a part of a system are unpredictable, we need to introduce some invariant property in order to be able to check for this invariance. Self-checking (S-C) design is used to achieve concurrent error detection using means of hardware redundancy. A complex circuit is partitioned into its element functional blocks and each of these blocks is implemented according to the structure of 2.1.

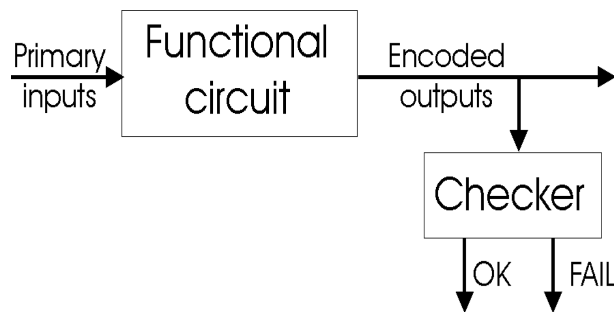

Figure 2.1: General structure of self-checking circuits

This structure implements functional blocks delivering outputs belonging to an error detecting code, and thus introduces an invariant property that can be checked concurrently. A checker monitoring this code performs the concurrent error detection.

The desirable goal to be achieved with self-checking circuits is often declared as Totally Self-Checking(TSC) goal. This goal requires that, under any modeled fault, the first erroneous output of the functional block is signaled on the outputs of the checker. To achieve this goal, some properties must be verified by the functional block and the checker.

2.2 Design of Functional Blocks

Concerning the functional block the following properties are required.

- **Fault Secure:** Under each modeled fault the produced erroneous outputs do not belong to the output code. The reason for this property is obvious; if an erroneous output belongs to the code, the error is not detected and TSC goal is lost. Thus, the fault secure property is the most important property required for the functional block. Another useful property is the self-testing one. This property guarantees that for each fault there is at least one input vector, occurring during the circuit normal operation, that detects it. In fact, this property avoids the existence of redundant faults. Such fault remain undetectable and could be combined with new faults occurring later in the circuits, resulting on multiple fault that could destroy the fault secure and the self-testing properties (totally self-checking property) offer the highest level of protection [12].

- **Self-Testing:** For each modeled fault there is an input vector occurring during normal operation that produces an output vector which do not belong to the code.

- **Totally Self-Checking:** The circuits is both fault secure and self-testing.

  With the fault secure property it is guaranteed that a first fault always generates detectable errors. Then, assuming that between the occurrence of two faults a sufficient time elapses so that the functional block receives all inputs required to test its faults (i.e., sufficiently long MTBF), the self-testing property guaranties that the first fault is detected before a second fault occurs in the S-C system. This way the TSC goal is achieved for a TSC functional block.

  The TSC property can be generalized into the strongly fault secure property which defines the largest class of functional circuits achieving the TSC goal.

  The fault secure property is the most important one since it guaranties error detection under any single fault, but it is also the most difficult to achieve. The self-testing property can be easily achieved, especially for stuck-at faults, where it is enough to remove the redundant faults by simplifying the circuit. Concerning the fault secure property, the most obviously way for achieving it is to duplicate the functional block and use a comparator to check for equality the delivered outputs. Since this solution appeals for a hardware cost higher than 100%, more complicated techniques are developed for reducing this cost. These techniques use error detection codes with cost lower than the duplication code.

2.3 Fail-Safe Design

The last stage of an electronic system often drives some actuators for controlling elements of the external world. Many systems have states that can be considered as safe. That is, they do not involve catastrophic events if they occur erroneously. A typical safe state is for instance the red color in traffic lights. In safety critical applications, each actuator must be controlled by a fail-safe signal, (i.e., a signal which in presence of failures is either correct or safe. Self-checking system deliver groups of encoded signals and are not adequate for driving these actuator (since each actuator is controlled by a single line which must be fail-safe individually). Due to this particular requirement it was not possible to implement fail-safe systems in VLSI. Therefore, existing fail-safe systems are composed of a self-checking or fault tolerant processing system (e.g., using error

detecting codes, duplication, triplication etc.), and of a fail-safe interface implemented using specific discrete components with very low probability to fail into the non safe direction.

This interface transforms the outputs of the processing system into fail-safe signals. The drawback of these interfaces is that they are very cumbersome and have a high cost. Furthermore, using discrete components result on lower MTTF with respect to VLSI implementations, and system availability is reduced. It is therefore mandatory to provide more compact fail-safe interfaces. However, few results have been published in this domain.

2.4 Error detection codes

Concurrent fault detection circuits (CFDCs) are essential components for on-line testing in systems designed for high reliability, high availability, and designed to be diagnosable to a replaceable unit such as a PCB or chip. CFDCs are also referred to as concurrent error detection (CED) circuits. CFDCs are typically incorporated in VLSI and PCB designs to support system-level fault recovery and maintenance strategies. These circuits are typically applied to ASIC design in an ad hoc manner and usually only to the data path circuits. Extensive use of CFDCs has been made in many digital systems such as electronic switching systems.

There are many types of error detecting codes (EDCs) and error correcting codes (ECCs) used in the design of CFDCs. The different types of CFDCs require varying degrees of information redundancy (extra bits) in the system circuitry for the EDC/ECC code word. However, not all of these codes are useful in practical system applications because of the large area and performance penalties associated with their hardware implementation. Therefor, the choice of a CFDC has considerable impact on the overall area and cost of the final system.

Unlike fault tolerant hardware structures that use hardware redundancy such as N-tuple Modular Redundancy (NMR), CFDCs are based on information redundancy using EDCs or ECCs. While there are many types of EDCs and ECCs, not all of these are useful in practical system applications because area and performance penalties that result from the circuitry required to generate the code words. Code word generation is preformed on the data at the data source before entering the CUT and the code words are checked (which requires regeneration of the code word) at the output of the CUT. Partitioning a system into sub circuits and inserting the code word check and regeneration circuits to detect faults at intermediate points facilitates effective fault isolation and diagnosis of replaceable units.

2.4.1 Parity code

It detects all single errors and more generally all errors of odd multiplicity. It is the cheapest code since it adds only one check bit to the information part. This check bit is computed to make constant the parity of each code word. As a matter of fact we can use an odd parity code (odd number of 1's in each code word) or an even parity code (even number of 1's in each code word).

2.4.2 Dual-Rail code

It is a variety of the duplication code where the check bits are equal to the complements of the information bits. This code has very strong error detection capabilities since it detects any errors affecting either the information part or its complement. But of course it is quite expensive since it duplicates the information.

2.4.3 M-out-of-n code

This code is a non-separable code (information and check bits are merged). It is composed by code words that have exactly m 1's, (e.g., 2-out-of-4 code: 1100, 1010, 1001 etc.). This code is an optimal non-separable unordered code (minimal redundancy for unordered coding).

2.4.4 Berger code

Berger code [14] can detects all multiple unidirectional errors (where the bits in error fail as logic 1s or logic 0s but not both in the same data word) but provides no error correction capability.

The basic idea is to count the number of logic 1s in the data word and use the inverted binary count value as the code word. By inverting the count value for use as the code word, we are able to detect a stuck-at-0 fault on a serial data line since the Berger code bits would be all 1s to indicate an all 0s data word. The number of data bits to be serviced by the Berger code word can be variable but should be less than (2^N), where N is the number of Berger code bits, to ensure optimal error detection.

2.4.5 Arithmetic codes

These codes are divided into separable and non-separable. In separable arithmetic codes of base A, the code words are obtained by associating to an information part X a check part X' equal to the modulo A of the information part, that is $X' = |X|_A$ (residue code), or $X' = A - |X|_A$ (inverse residue code).

In non-separable arithmetic codes of base A (or AN codes), the code words are equal to the product of the original (i.e., non-coded) word by the base A.

Arithmetic codes [12] are interesting for checking arithmetic operations, because they are preserved under such operations. The most useful arithmetic codes are the separable ones, and they are most often implemented as low cost arithmetic codes, where the check base A is equal to $2^m - 1$. In this case a m-bit modulo A adder is realized by a m-bit adder having the carry-out signal feeding back the carry-in signal (carry end-around adder). Then, the check part generator for the low cost arithmetic codes is realized as a modular network using these adders as building blocks. Low cost arithmetic codes detect variable arithmetic errors according to the value of the check base.

2.4.6 Hamming code

Hamming code [14] provides not only error detection but also error correction capability based on an extension of the principles of parity. The Hamming code word is constructed

from the parity bits of various combinations of the data bits determined by the parity check matrix. Note that the decimal value of each bit position in the parity check matrix corresponds to the binary value of the parity check matrix. Also note that the Hamming code bits, H_i,occupy 2^n positions in the parity check matrix and ,as a result, have only a single 1 in any position in the column below the H_i. It is easy to extend this matrix to accommodate any desired size data word with a new Hamming code bit introduced each time a new 2^n value position is encountered. Each Hamming code bit is generated by the exclusive-OR of all the data bits, D_i, that have a 1 in the same row at the corresponding Hamming bit.

Hamming circuits are more complex than parity circuits in terms of the number of gates and the number of additional bits required for the code word. However, Hamming code is quite efficient compared to other error correcting codes in terms of the area overhead and performance penalty required for the error correction process. The Hamming circuit models are based on single-bit error-correcting parity codes which use M Hamming bits to detect and correct single-bit errors in N data bits. Given N data bits, the required value of M can be calculated by the relationship $2^M > M + N$. If a single bit error is detected, the error can be corrected in the output data bus and the presence is indicated by the active error signal.

2.5 Design of Checkers

The mission of a checker is to signal the occurrence of a code input (by generating on its output a correct operation indication), and the occurrence of a noncode input (by generation an error indication. The set of output words indicating correct form the output code space of the checker and the set of output words indicating error occurrence form the output non code space. As an implication of this mission the checker verifies the code disjoint property [12].

2.5.1 Code-Disjoint

The checker maps code inputs into code outputs and noncode inputs into noncode outputs.

Code-disjointness is not related to the testability of the checker. It simply reflect a functional property. However a fault occurring in the checker may alter its ability to produce an error indication output under a noncode input. If this fault is not detected, another fault can later occur in the functional block. Then, an erroneous noncode output produced by this block eventually will not be signaled by the checker due to its proper fault. To cope with this problem, the checker must verify the self-testing property.

2.5.2 Self-Testing

For each modeled fault there is a code input that produces a noncode output.

As for functional blocks, assuming a sufficiently long Mean Time Between Failures (MTBF), the self-testing property guaranties that the fault is detected before the occurrence of another fault in the system. This way the TSC goal is achieved.

The self-testing code-disjoint checkers can be generalized into the strongly code-disjoint checkers, which define the largest class of checkers allowing to achieve the TSC goal.

The design of self-testing checkers is a difficult task because it requires to detect all the faults in the checker by applying only code inputs. Fortunately, we have to consider a limited number of checkers classes corresponding to the more useful error detecting codes. For these checkers, extensive investigations by numerous researches have accomplished this task. Thus, there are today available self-testing checkers for all the error detecting codes used in self-checking design.

A first important implication of the self-testing property is that a checker must have at least two outputs. In a single-output checker, the one output value (e.g., logic 0) must be used for correct operation indication and the second (e.g., logic 1) for error indication. Then, a stuck-at on the value corresponding to the correct operation indication can not be detected and the checker is non self-testing. Such a fault is very dangerous since it will mask any subsequent fault occurring in the functional block. Because of this situation it is generally adopted in the self-checking community the use of two output checkers. The dual-rail indication, while the values 00 and 11 are used for error indication.

2.6 Perturbation Tolerant Memories

Complex electronic systems are subject to transient faults provoked by various causes such as electromagnetic interference, cross-talk, alpha particles, cosmic rays etc. Transients represent the main cause of failures in complex electronic systems. In some particular applications, like space for instance, protection against soft errors) single event upset (SEUs) caused by heavy ion strikes) is mandatory. Strong requirements for protection against transients also exist in fault tolerant systems and in safety critical applications. Also, the introduction of deep submicron technologies increases significantly the sensitivity of VLSI circuits to the various causes of transients. As a matter of fact hardware techniques for designing perturbation tolerant circuits may have a considerable impact on the design of a large number of electronics systems [12].

Memory elements represent the most sensitive parts of a CMOS circuit, since static CMOS logic is drastically less sensitive than the memory cells with respect to the various causes of transient faults. Thus, perturbation resistant/tolerant memory design is the key point for designing perturbation tolerant ICs. Perturbation tolerant design for large memory arrays (e.g., large RAMs, caches, etc.) can be achieved efficiently by means of error correcting codes. However, this solution can not be used in the case of memory elements distributed across the logic of an IC, but also it is very expensive for implementing small embedded memories for which the cost of an error correcting code (check bits plus the error correction controller) will be very high. In these situations using perturbation hardened memory cells are the most appropriate alternative.

Chapter 3

Related work

Previous approaches benefit from the fact that it was possible to work at functional/logical level, by providing the necessary fault observability properties to each node constituting the functional description of the device under consideration. With ASIC, even when mapping with different technological libraries, commercial tools are able to maintain the functional description of each node constituting the network, thus producing the TSC device even when the used gates are not exactly the ones identified by the Boolean equations. With FPGA, nodes constituting the network are being collapsed and merged to better suit the basic CLB elements constituting the FPGA resources in order to minimize the used area. The operation modifies each fault observability, thus potentially not fulfilling the required and previously provided fault-error relation, hence no assumptions can be made on the observability of each fault on the primary outputs, so that a subsequent TSC fault analysis and re-design steps are necessary.

New approach was presented in [4]. This paper addresses the issue of self-checking FPGA design based on the adoption of error detection codes (e.g., Berger code, Parity code) as an evolution of the traditional approaches developed in the past years for the ASIC platform. They investigated the applicability of design techniques defined for introducing hardware fault detection properties in a combinational network through information redundancy at functional/gate level. This approach is the starting point for the definition of a more complete methodology to dynamically reconfigure FPGAs in response to a fault, once it has been detected. Furthermore, they was presently adapting the original fault-error analysis tool to work on the circuit description produced by the Leonardo, so that the fault-error relation enforcement can be directly suited for the FPGA thus better controlling the effects of commercial tools' manipulations and the presence of unused logic.

The goal of the proposed investigation is to explore the suitability of Concurrent Error Detection (CED) techniques based on Error Detection Codes for the FPGA platform. Given this premise, the attention has been initially devoted to the stuck-at faults and to single upset events (SEU) that may corrupt the internal memory or the LUTs.

Another approach focusing on CED techniques using hardware redundancy is presented in [1] [2]. Concurrent error detection (CED) techniques (based on hardware duplication, parity codes, etc.) are widely used to enhance system dependability. All CED techniques introduce some form of redundancy. Redundant systems are subject to common-mode failures (CMFs). While most of the studies of CED techniques focus on area overhead, few analyze the CMF vulnerability of these techniques. In this paper, for the first time,

we present simulation results to quantitatively compare various CED schemes based on their area overhead and the protection (data integrity) they provide against multiple failures and CMFs. Our results indicate that, for the simulated combinational logic circuits, although diverse duplex systems (with two different implementations of the same logic function) sometimes have marginally higher area overhead, they provide significant protection against multiple failures and CMFs compared to other CED techniques like parity prediction.

Concurrent Error Detection (CED) techniques are widely used to enhance system dependability. Almost all the CED techniques are based on the following principle: Let us suppose that the system under consideration realizes a function f and produces output f(i) in response to an input sequence i. A CED scheme generally contains another unit which independently predicts some special characteristic of the system-output f(i) for every input sequence i. Finally, a checker unit checks whether the special characteristic of the output actually produced by the system in response to input sequence i is the same as the one predicted and produces an error signal when a mismatch occurs. Some examples of the characteristics of f(i) are: f(i) itself, its parity, 1s count, 0s count, transition count, etc. The architecture of a general CED scheme is shown in Figure 3.1 . Any CED scheme is characterized by the class of failures in the presence of which the system data integrity is preserved. By data integrity, we mean that the system either produces correct outputs or indicates erroneous situations when incorrect outputs are produced. In the literature on fault-tolerance, this property has been referred to as the fault-secure property.

It may be noted that the general architecture of a CED scheme such as Figure 3.1 relies on the use of hardware redundancy (predictor and checker circuits) for error-detection. Time redundancy techniques like *alternate-data-retry* and *recomputation* with shifted operands can also be used for concurrent error detection. Time redundancy directly affects the system performance although the hardware cost is generally less than that of hardware redundancy.

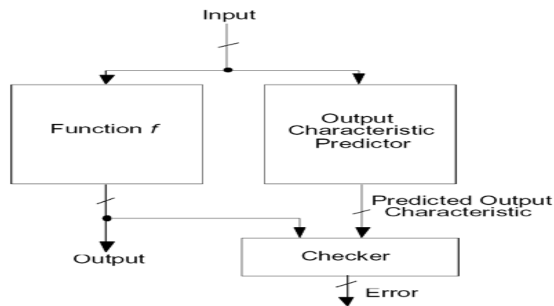

Figure 3.1: Used fault model

Several CED schemes for designing reliable computing systems have been proposed and used commercially. These techniques mainly differ in their error-detection capabilities and the constraints they impose on the system design. There are many publications on system design with concurrent error detection. These include designs of datapath circuits (like adders, multipliers, etc.), and general combinational and sequential logic circuits with concurrent error detection. Almost all publications on CED focus on their area/performance overhead. However, the systems considered are restricted to those with redundancy through replication. All the above-mentioned CED techniques guarantee

system data integrity against single faults. However, these CED schemes are vulnerable to multiple faults and common-mode failures. Common-mode failures are a special and very important cause of multiple faults.

Common-mode failures (CMFs) produce multiple faults, occurring generally due to a single cause; the system data integrity is not guaranteed in the presence of CMFs. These include design mistakes and operational failures that may be due to external (such as EMI, power-supply disturbances and radiation) or internal causes. CMFs in redundant VLSI systems are surveyed in [3]. Design diversity has been proposed in the past to protect redundant systems against common-mode failures.

Chapter 4

Overview of our approach

This report introduces a new method of generating totally self-checking (TSC) circuits based on error detecting codes. Our work is focused on combinational circuits with unknown inner structure. An independent method to generate TSC circuits play an important role for the design tools which help to simplify the implementation of such codes in a large design. The basic idea is to design an appropriate method to ensure 100% fault coverage. An area overhead plays an important role in the final TSC circuit.

The proposed method assumes that a complex design is divided into simple basic functional blocks. Each simple block is modified independently of other blocks. When the simple block fails the reconfiguration process [9][10][11] can be initiated by another TSC circuit.

We have used the structure on Figure 4.1 as a basic model of the totally self-checking circuit. The final scheme consists of three basic blocks: the original combinational circuit, the check bits generator and the checker. The checking bits are generated from the primary inputs of the original circuit. The primary output and the checking bits are used as an encoded output. The checker compares the check bits with the check bits generated directly from the primary outputs. The condition of the self-checking property which has to be satisfied is that a checker must have at least two outputs [12]. The first output is used for regular operation and the second for error indication. This basic structure ensures that a circuit can be totally self-checking (TSC). Another condition which has to be satisfied is that the basic structure has to be self-testing and fault secure.

4.1 Error detecting codes

The error detecting codes are important for a self-checking design. Error detecting and correcting codes such as the SEC/DED Hamming codes are very useful in a context of error correction in memory systems and transmission channels. It is very important that the correcting codes cannot be used to correct error output in our case. This is due to the fact that we cannot say what is the correct word. In other words, if we correct the incorrect code word we obtain a wrong output.

The design of fault secure functional blocks is a difficult problem. For a given output code it must be guaranteed that under each modeled fault the produced erroneous outputs do not belong to the output code. The choice of the output code is a very critical task. The selected code that has high error detection capabilities can easier achieve the fault

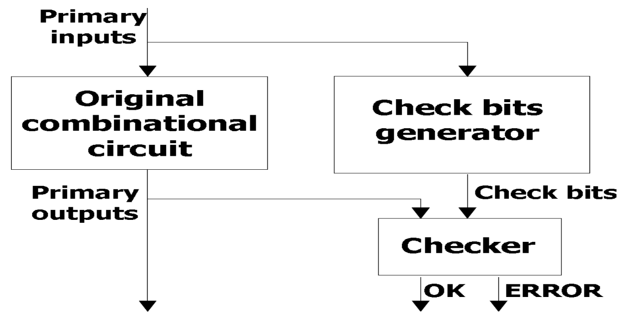

Figure 4.1: General architecture of a concurrent error detection scheme

secure property but adds a large number of outputs, thus, increasing hardware cost. On the other hand, selecting a code with low error detection capabilities will add fewer extra outputs but, for achieving fault secureness, it may require significant modification of the circuit structure (and thus extra cost). As a matter of fact, the selection of the output code is made by analyzing the particular circuit, in order to obtain the best result.

4.2 Fault model

All of our experiments use FPGA circuits. The combinational part consists of individual memory elements (LUTs - look up tables). In figure 4.2 we can see 3 gates mapped into a LUT. The original circuit has two inner nets. The original set of the test vectors covers faults on these inner nets too. For the look up table this test vectors are redundant. For circuits realized with look up tables the change in the memory leads to a stuck-at fault at the primary output of the look up table. We have used the stuck-at fault model in our experiments.

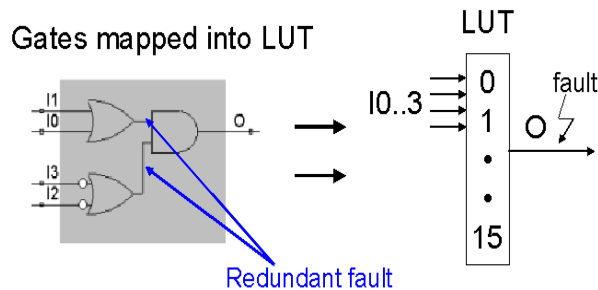

Figure 4.2: Used fault model

The used fault model is described by a simple example in Figure 4.3. For simplicity we have used only one LUT. The LUT realizes a part containing 3 gates. Primary inputs from I0 to I1 are the same as the address inputs for the LUT. We select an address, and then the value at this address is propagated to the output. We assume the following situation: The content of this LUT can be changed, e.g., an electromagnetic interference, cross-talk or alpha particles. The appropriated memory cell is set to one and the wrong value is propagated to the output. It means that the realized function is changed and

output behaves as a stuck at 1. From this example we can say that a change of any LUT cell leads to a stuck at fault on the output. This fault is observed only if the bad cell is selected. This is the same for circuits based on gates. Some fault can be masked and does not necessarily lead to an erroneous output.

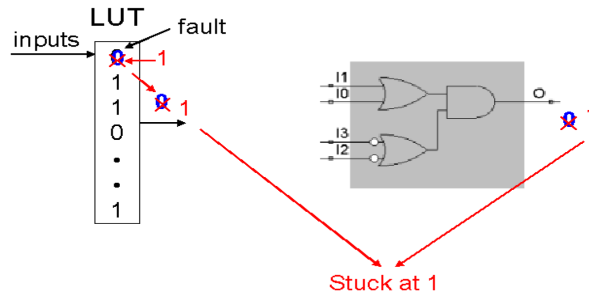

Figure 4.3: Example of used fault model

4.3 Basic idea

The basic idea how to ensure self-checking circuit is to add a combinational circuit, which computes the output check bits from the primary inputs. The solution is shown in Figure 4.4. There is the original circuit, its primary inputs and primary outputs. To obtain the checking bits we duplicate the original circuit and add the code generator circuit. By this way we obtain the code word.

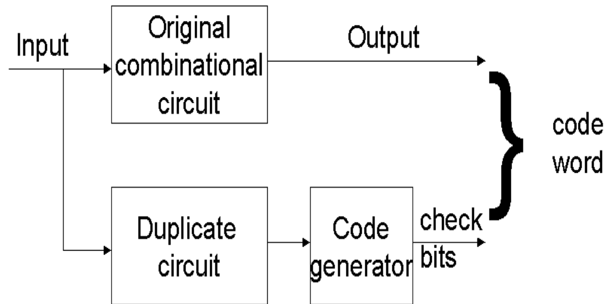

Figure 4.4: Basic idea of check bits generator

4.4 Hamming code

The Hamming code is defined by its generating matrix [13]. For simplicity we use the matrix containing the unit submatrix at the right hand. The generating matrix of Hamming code (15,11) is shown in Figure 4.1. The values a_{ij} have to be defined.

When a more complex Hamming code is used, more values have to be defined. The number of outputs o_i used for checking bits determines the appropriate code. For example

$$\left(\begin{array}{cccc|cccc} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & 1 & 0 & \cdots & 0 \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & 0 \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & 0 & 0 & \cdots & 1 \end{array}\right)$$

Table 4.1: Generating matrix for Hamming code (15,11)

the circuit c432 having 7 outputs requires at least Hamming code (15,11). In this case we use 7 data bits and 4 checking bits. The definition of values a_{ik} is also important.

~~0000~~	~~1000~~
~~0001~~	1001
~~0010~~	1010
0011	1011
~~0100~~	1100
0101	1101
0110	1110
0111	1111

Figure 4.5: Generating left part of matrix

Now we present a method of generating values a_{ik}. Let us mention the Hamming code (15,11) having 4 checking bits. The generating of left side of the generating matrix can be divided into 3 basic steps (see Figure 4.5):

- We generate all words of n bit binary code (2^n words). n means the number of check bits. For the Hamming code (15,11), n is equal to 4.
- We remove the zero word.
- We remove all words containing only one 1.

Now we got 11 code words. All of these words are used to fill the left side of the generating matrix. The number of vectors in the set is the same as the number of rows in an appropriate Hamming code. This method ensures that vectors are equal and due to the fact that there are no zero rows and rows with only one 1 we have got the best solution to fill right side of the matrix. Then we generate the circuit for checking bits x_k

$$x_k = a_{1k}o_1 \oplus a_{2k}o_2 \oplus \cdots a_{mk}o_m \tag{4.1}$$

where $o_1 \cdots o_m$ are outputs of the circuit.

A problem can occur when we use fewer outputs than the matrix was generated for (shortened Hamming code, see Figure 4.6). For example we assume only 4 data outputs. The left part of this string consists only of zero bits. Zero bits indicate that the data bits are not used to compute the check bit.

On the other hand 1 bits in the string means, that the data outputs are used for the calculation of the check bit. We have a problem with no 1 in row. Scattering will provide a solution to this problem.

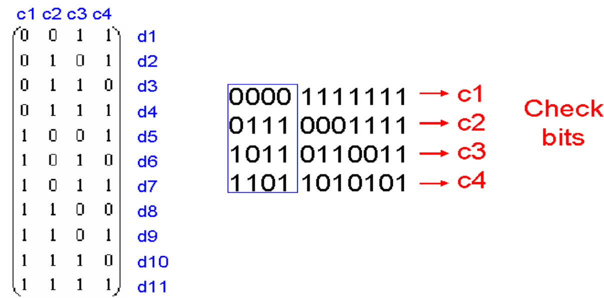

Figure 4.6: Problem with Matrix Regularity

The Figure 4.7 shows the original left side of the generating matrix. Now we take the odd rows and we rewrite them to the same position in the new matrix. Next we rewrite even rows into the new matrix. We use the rule shown as arrows in the Figure 4.7.

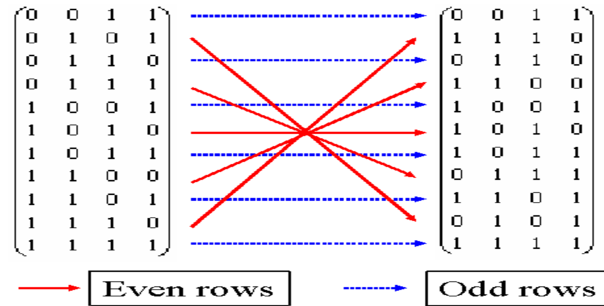

Figure 4.7: Scattering - Example

4.5 Design scheduling

Figure 4.8 describes how we make a test for every detecting code. As an input we have used the ISCAS85 benchmarks [20]. Every benchmark is duplicated by renaming the original circuit and loading the original circuit again. Next we add the code generator behind the duplicated part. Now we generate the modified benchmark. The Atalanta ATPG tool was used to generate the minimal tests for benchmarks. The obtained test set with the modified benchmark is put into the last part. In this part we inject a fault and by our simulator we compute the fault coverage. The bold rectangles represent our original software.

4.6 Software solution

The first task is the modification of the original circuits. It means that this part ensures loading, saving and renaming a circuit. To duplicate a circuit we have to read the original circuit, rename this circuit and load the original circuit again. The second task is adding the check nets. This could be done by one of these methods: by adding a net or by adding

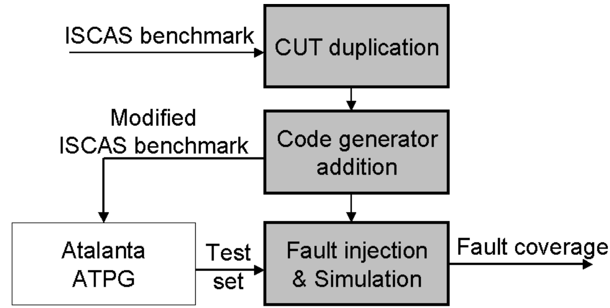

Figure 4.8: Design scheduling of self-checking circuit (dark boxes represent our software).

a gate. The third task is a simulation of the modified circuit and the fault injection. The last task is the computation of the fault coverage for the whole circuit (the original circuit and the check bits generator). All the software was written in Microsoft Visual C++. Because our software cannot generate the test set, we have used the Atalanta ATPG tool.

4.7 Check bits generator

The area of the check bits generator contributes significantly to the total area of the TSC circuit. As an example we consider circuit with 3 inputs (c,b and a) and 2 outputs (f and e). The check bits generator uses the odd parity code to generate check bits. In our example we have only one check bit x.

cba	fex
000	010
001	100
010	100
011	100
100	010
101	010
110	111
111	001

Figure 4.9: Table

Our example is shown in Figure 4.9. The output x was calculated from two output e and f. At this time we have to generate the minimal form of the equation. The minimal form we can achieve with methods like the Karnaugh map or Quine-McCluskey. After minimization we got three equations, one per every output (f, e and x), where x means odd parity of the outputs f and e. If we want to know if the odd parity covers all faults in our example of simple combinational circuit we have to generate the minimal test set and simulate all faults on every net in this circuit.

Final equations are:

$$e = \overline{b}c + \overline{a}(\overline{b} + c) \tag{4.2}$$

$$f = \overline{a}b + \overline{c}(a + b) \tag{4.3}$$

$$x = bc \tag{4.4}$$

Chapter 5

Experimental results

5.1 Used benchmarks

All our experiments use the ISCAS85 benchmarks [20] where all the circuits are combinational only. These benchmarks are based on real circuits from large designs.

Description of tested benchmarks:

- **c432** 27-channel interrupt controller
- **c499/c1355** 32-bit SEC circuit
- **c880** 8-bit ALU
- **c1908** 16-bit SEC/DED circuit
- **c2670** 12-bit ALU and controller

5.1.1 Even Parity

The even parity, which is the simplest checking code, was used for our first experiment. The results, presented in Table 5.1, show that one parity bit cannot cover all the faults inserted into the tested circuit. The circuit c17 was not used for our experiments because of its simplicity.

Table 5.1: Application of even parity code

Circuit	Inputs	Outputs	Redundancy parity bits	All tested faults	Detected faults	Detected faults[%]	Area occupation[LUT]	
							Redundancy part	Data part
c432	36	7	1	5536	4626	83,56	72	69
c499	41	32	1	15150	14628	96,55	87	88
c880	60	26	1	24567	23998	97,68	114	112
c1355	41	32	1	64165	62472	97,36	92	87
c1908	33	25	1	134012	119280	89,01	125	120
c2670	233	140	1	105532	84840	80,39	166	175

5.1.2 Double even Parity

The double parity was used to generate the checking bits as the second experiment. Even and odd bits of outputs are coded separately by even parity. The results of this experiment are presented in Table 5.2.

Table 5.2: Application of double even parity code

Circuit	Inputs	Outputs	Redundancy parity bits	All tested faults	Detected faults	Detected faults[%]	Area occupation[LUT] Redundancy part	Data part
c432	36	7	2	5433	5012	92,25	73	69
c499	41	32	2	13984	13750	98,33	99	88
c880	60	26	2	27495	27206	98,95	120	112
c1355	41	32	2	62834	62012	98,69	90	87
c1908	33	25	2	130140	124958	96,02	124	120
c2670	233	140	2	116220	111270	95,74	219	175

5.1.3 Hamming code (63,57)

Both previous experiments (5.1, 5.2) did not reach 100% fault coverage of the tested circuits. The third experiment is based on the Hamming code (63,57), where the maximum number of data bits is 57 and the number of checking bits is 6. Experimental results are shown in Table 5.3. The fault coverage for c499 and c1355 benchmarks is 100%. It means that the Hamming code (63,57) is appropriate with respect to the maximal fault coverage. We must mention that the fault coverage depends on the minimal testing set. If the minimal test created by Atalanta software does not cover all faults, we cannot say that simulated circuits are 100% fault covered. In other words some faults cannot be detected because the minimal test set does not cover the redundant faults. This Hamming code cannot be used for benchmark c2670 because the number of its outputs is bigger than the Hamming code can cover.

Table 5.3: Application of Hamming code(63,57)

Circuit	Inputs	Outputs	Redundancy parity bits	All tested faults	Detected faults	Detected faults[%]	Area occupation[LUT] Redundancy part	Data part
c432	36	7	6	5569	5544	99,55	77	69
c499	41	32	6	17791	17791	100,00	116	88
c880	60	26	6	27109	27106	99,99	140	112
c1355	41	32	6	68647	68647	100,00	117	87
c1908	33	25	6	123651	123376	99,78	145	120

5.1.4 Hamming code (255,247)

The fourth experiment is based on the Hamming code (255,247). The maximum number of data bits is 247 and the number of checking bits is 8. In our case only 7 outputs are used. The experimental results are shown in Table 5.4.

Table 5.4: Application of Hamming code(255,247)

Circuit	Inputs	Outputs	Redundancy parity bits	All tested faults	Detected faults	Detected faults[%]	Area occupation[LUT]	
							Redundancy part	Data part
c432	36	7	7	5694	5602	98,38	74	69
c499	41	32	7	18003	18003	100,00	111	88
c880	60	26	7	30277	30277	100,00	134	112
c1355	41	32	7	69634	69634	100,00	104	87
c1908	33	25	7	135402	134600	99,41	138	120
c2670	233	140	7	160092	160061	99,98	314	175

5.1.5 Partial conclusion

We can summarize, that all of our experiments say that 100% fault coverage can be reached using more redundancy outputs generated by special codes. The Hamming code can be used as a suitable code to generate parity bits. Its type depends on the number of outputs and on the complexity of the original circuits. It means that more complex circuits need more parity outputs.

Chapter 6

Conclusions and future work

This work is a part of the methodology of automatic design of concurrent error detection (CED) circuits based on FPGA with the possibility of dynamical reconfiguration of the faulty part. The reliability characteristics can be increased by reconfiguration after error detection. The most important criterium is the speed of the fault detection and the safety of the whole circuit with respect to the surrounding environment. We can summarize, all of our experiments say that 100% fault coverage can be reached for whole design including checking parts. It is achieved by using more redundancy outputs generated by special codes. The Hamming code can be used as a suitable code to generate check bits. Its type depends on the number of outputs and on the complexity of the original circuit. More complex circuits need more check bits. We would like to reduce the duplicated circuit and compute the fault coverage again. We have proposed a new solution of creating the check bits generator. Because we want to increase the reliability characteristics of the circuit implemented in FPGA we have to modify the circuits at the netlist level. The implemented design has to satisfy the condition of modularity. Due to this fact we have proposed a special design [2] suitable for TSC and reconfiguration properties. The TSC property must be fulfilled for every modules and of course for the whole design too. We have proposed such structure that satisfies self-checking properties and enables dynamical reconfiguration, see Figure 6.1. The number of outer nets and the complexity of every block affects the fault coverage and the final area overhead.

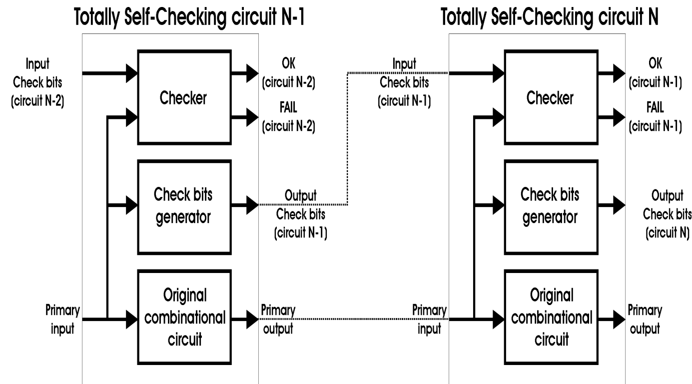

Figure 6.1: Proposed structure of TSC circuits implemented in FPGA

All of our experiments involve the combinational circuits, but many circuits in real

designs are composed of sequential parts. Our solution will divide the original circuit into simple combinational parts separated by flip-flops. As an example, the finite state machine can be divided into two parts, where the first part covers combinational logic from inputs to flip-flops (with feedback) and the second one covers the combinational logic from flip-flops to outputs (with the nets that are connected directly from the input to the output).

Bibliography

[1] S. Mitra and E. J. McCluskey. *Which Concurrent Error Detection Scheme To Choose?* Proc. International Test Conf., pp. 985-994, 2000.

[2] S. Mitra and E. J. McCluskey. *Diversity Techniques for Concurrent Error Detection* Center for Reliable Computing, Dept. of Electrical Engineering and Computer Science Stanford University

[3] S. Mitra, N. R. Saxena and E. J. McCluskey, *Common-Mode Failures in Redundant VLSI Systems: A Survey* IEEE Trans. Reliability, 2000,

[4] C. Bolchini, F. Salice and D. Sciuto. *Designing Self-Checking FPGAs through Error Detection Codes.* 17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'02), November 06 - 08, 2002 , Canada.

[5] K. Elshafey and J. Hlavička. *On-Line Detection and Location of Faulty CLBs in FPGA- Based Systems.* IEEE DDECS Workshop, Brno, Czech republic, April 17-19, 2002, pp. 183-190.

[6] Xilinx Corp. *XAPP 151 (v1.5), Virtex Series Configuration Architecture User Guide.* Xilinx Corp., 2000.

[7] M. Abramovici, C. Stroud, C. Hamiliton, S. Wijesuriya and V. Verma. *Using Roving STARs for On-Line Testing and Diagnosis of FPGAs in Fault-Tolerant Applications.* Proceeding IEEE International Test Conference, pp. 973-982, 1999.

[8] H.K. Lee and D.S. Ha. *Atalanta: an Efficient ATPG for Combinational Circuits.* Technical Report, 93-12, Dep't of Electrical Eng., Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1993.

[9] Atmel Corp. *AT40K Series Configuration.* Atmel Corp., 2002.

[10] Xilinx Corp. *XAPP 138 (v2.5) Virtex FPGA Series Configuration and Readback.* Xilinx Corp., 2001.

[11] Xilinx Corp. *XAPP 153 (v1.0), Status and Control Semaphore Registers Using Partial Reconfiguration.* Xilinx Corp., 1999.

[12] M. Nicolaidis and Y. Zorian. *On-Line Testing for VLSI - A Compendium of Approaches.* On-Line Testing for VLSI, Kluwer Academic Publisher, London 1998, ISBN 0-7923-8132-7.

[13] J. Adámek. *Foundations of coding.* A Wiley-Interscience Publication, JOHN WILEY & SONS, INC, United States of America 1991, ISBN 0-471-62187-0.

[14] Ch. E. Stroud. *A Designer's Guide to Built-In Self-Test.* Kluwer Academic Publisher, London 2002, ISBN 1-4020-7050-0.

[15] M. L. Bushnell and V. D. Agrawal. *Essentials of Electronic Testing.* Kluwer Academic Publisher, London 2000, ISBN 0-7923-7991-8.

[16] D. K. Pradhan. *Fault-Tolerant Cumputer System Design.* Prentice Hall PTR, Upper Saddle River, New Jersey 1996, ISBN 0-7923-7991-8.

[17] A. Paschalis, D. Gizopoulos, N. Gaitanis. *Concurrent Delay Testing in Totally Self-Checking System.* On-Line Testing for VLSI, Kluwer Academic Publisher, London, 1998.

[18] S. J. Piestrak. *Design of Self-Testing Checkers for m-out-of-n Codes Using Parallel Counters.* On-Line Testing for VLSI, Kluwer Academic Publisher, London, 1998.

[19] D. Nikolos. *Self-Testing Embedded Two-Rail Checkers.* On-Line Testing for VLSI, Kluwer Academic Publisher, London, 1998.

[20] F. Brglez, H. Fujiwara. *A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortan.* Proc. of International Symposium on Circuits and Systems, pp. 663-698, 1985.

Chapter 7

Dissertation thesis

Title: Design of Self Checking Circuits Implemented in FPGA

Abstract

Dissertation thesis will focus on the methodology of automatic design of totally self-checking circuits (to achieve better reliability characteristics by partial dynamic reconfiguration). The following topics will be investigated:

- The check bits generator will be optimized in order to obtain minimal area overhead.
- The existing solution for combinational circuits will be extended to sequential circuits.
- The TSCs design methodology will be extended from individual modules to large designs with consideration of further dynamic reconfiguration (see Figure 6.1).

My proposed methodology will be verified by ISCAS85 benchmarks implemented in FPGA.

Keywords

on-line testing, self-checking circuits, fail-safe circuits, error detecting codes, FPGA, dynamic reconfiguration

Chapter 8

Publications of the author

[A1] P. Kubalík and J. Buček. *FPGA Implementation of USB 1.1 Device Core.* In: Proceedings of Workshop 2003 (web) [CD-ROM]. Prague : CTU, 2003, vol. A, p. 304-305. ISBN 80-01-02708-2.

[A2] P. Kubalík and J. Buček. *FPGA Implementation of USB 1.1 Device Core.* In: Poster 2003. Prague : CTU, Faculty of Electrical Engineering, 2003, p. IC22.

[A3] P. Kubalík and H. Kubátová. *Design of Self Checking Circuits Based on FPGA.* In: Proceedings of the 15th International Conference on Microelectronics. Cairo : Cairo University, 2003, p. 378-381. ISBN 977-05-2010-1.