McMaster University
DigitalCommons@McMaster

Open Access Dissertations and Theses | Open Dissertations and Theses

Fall 2012

# A Formal Approach to Concurrent Error Detection in FPGA LUTs

Jameson P. Bergstra
*McMaster University*, bergstjp@mcmaster.ca

Follow this and additional works at: http://digitalcommons.mcmaster.ca/opendissertations

Part of the Digital Circuits Commons

Recommended Citation
Bergstra, Jameson P., "A Formal Approach to Concurrent Error Detection in FPGA LUTs" (2012). *Open Access Dissertations and Theses.* Paper 7431.

This Thesis is brought to you for free and open access by the Open Dissertations and Theses at DigitalCommons@McMaster. It has been accepted for inclusion in Open Access Dissertations and Theses by an authorized administrator of DigitalCommons@McMaster. For more information, please contact scom@mcmaster.ca.

# A Formal Approach to Concurrent Error Detection in FPGA LUTs

# A FORMAL APPROACH TO CONCURRENT ERROR DETECTION IN FPGA LUTS

BY

PETER BERGSTRA, B.Eng.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

© Copyright by Peter Bergstra, July 2012

All Rights Reserved

Master of Applied Science (2012) McMaster University

(Computing and Software) Hamilton, Ontario, Canada

TITLE: A Formal Approach to Concurrent Error Detection in FPGA LUTs

AUTHOR: Peter Bergstra

B.Eng., Mechatronics Engineering

McMaster University, Canada

SUPERVISOR: Dr. Mark Lawford and Dr. Nicola Nicolici

NUMBER OF PAGES: x, 83

# Abstract

In this thesis we discuss a formal approach to the design of concurrent error detection (CED) logic in field-programmable gate arrays (FPGAs). Single event upsets (SEUs) occurring in look-up table (LUT) configuration bits are considered as the fault model. Our approach involves representing the LUT network of the design implemented in the FPGA with constraints to model the presence of SEUs as a boolean formula in conjunctive normal form. A quantified boolean formula (QBF) based approach to designing CED logic based on parity check codes is found to be infeasible for designs of a realistic size. It is shown that a satisfiability (SAT) solver can be used to find variable assignments that indicate which circuit outputs can be corrupted by upset events in the specified fault model. An algorithm is presented to automatically generate a parity check code, which will identify with one clock cycle detection latency a malfunction caused by an SEU. The resulting parity check logic can be verified using a SAT solver and it is shown to require fewer LUT resources than duplication for most circuits.

# Notation and abbreviations

| | |
|---|---|
| CED | Concurrent error detection |
| CNF | Conjunctive normal form |
| CPU | Central processing unit Cyclic redundancy check |
| CRC | Directed acyclic graph |
| DAG | Field-programmable gate array |
| FPGA | Fault secure |
| FS | Hardware description language |
| HDL | Logic array block |
| LAB | Logic element |
| LE | Literals |
| Lits | Look-up table |
| LUT | Path fault secure |
| PFS | Quantified boolean formula |
| QBF | Satisfiability |
| SAT | Structure-constrained logic optimization |
| SCLO | Single event upset |
| SEU | Strongly fault secure |
| SFS | Static random-access memory |
| SRAM | Triple-modular redundancy |
| TMR | Totally self-checking |
| TSC | Verilog Quartus mapping |
| VQM | |

# Contents

**Abstract** **iii**

**Notation and abbreviations** **iv**

**1 Introduction and Problem Statement** **1**

1.1 Safety-Critical Systems . . . . . 1

1.2 Field-Programmable Gate Arrays . . . . . 2

1.3 Single Event Upsets . . . . . 4

1.4 Problem Statement . . . . . 5

1.4.1 Fault Model . . . . . 5

1.4.2 Robustness Requirement . . . . . 6

1.4.3 Cost Requirement . . . . . 6

1.5 Contributions and Thesis Organization . . . . . 7

**2 Background and Prior Work** **9**

2.1 Boolean Satisfiability . . . . . 10

2.1.1 SAT Solvers . . . . . 11

2.1.2 QBF Solvers . . . . . 11

2.2 FPGAs . . . . . 12

2.2.1 Architecture Overview . . . . . 12
2.2.2 Look-up Tables . . . . . 13
2.2.3 Tool Flow . . . . . 14
2.3 Fault Tolerance in FPGAs . . . . . 15
2.4 The Totally Self-Checking Goal . . . . . 16
2.5 Concurrent Error Detection . . . . . 20
2.5.1 Berger Codes . . . . . 21
2.5.2 Bose-Lin Codes . . . . . 23
2.5.3 Parity Check Codes . . . . . 24
2.5.4 FPGA-based Approaches . . . . . 27
2.6 Summary of Prior Work . . . . . 28

**3 Formal Approach to CED 30**

3.1 Introduction . . . . . 30
3.2 CNF Formula of a Circuit . . . . . 32
3.2.1 Look-up Tables with SEUs . . . . . 33
3.2.2 Flip-Flops . . . . . 37
3.2.3 Other FPGA Resources . . . . . 39
3.3 Designing a Parity Check Matrix using QBF . . . . . 39
3.3.1 Representing the Parity Predict Logic and the Checker . . . . 40
3.3.2 Representing Fault Secure Constraints . . . . . 43
3.3.3 Finding a Parity Check Matrix . . . . . 44
3.3.4 Feasibility of Using QBF . . . . . 44
3.4 SAT-based Approaches . . . . . 45
3.4.1 Fault Secure Constraints for SAT . . . . . 45

3.4.2 Uses of SAT . . . . . 46
3.5 Proposed Procedure . . . . . 47
3.5.1 Run Time Improvements . . . . . 48
3.5.2 Extra Circuit Outputs . . . . . 48
3.5.3 Duplicating Internal Signals . . . . . 51
3.6 Conclusion . . . . . 51

**4 Selecting a Parity Check Matrix 53**
4.1 General Approach and Rationale . . . . . 54
4.1.1 Design Flow Benefits and Limitations . . . . . 55
4.1.2 Resource Estimation . . . . . 56
4.1.3 Logic Sharing Constraints . . . . . 58
4.1.4 Essential Outputs . . . . . 60
4.1.5 Algorithm Overview . . . . . 60
4.2 Parity Check Matrix Algorithm . . . . . 62
4.2.1 `AREA(i,j)` . . . . . 62
4.2.2 `SHARED(i,j)` . . . . . 63
4.2.3 `CHANGE(i,j)` . . . . . 65
4.2.4 `PARITY_CHECK` . . . . . 65

**5 Presentation of Results 68**
5.1 Benchmark Results . . . . . 68
5.2 Comparison with Prior Techniques Using Parity Check Codes . . . . 70
5.3 Microprocessor Circuit Results . . . . . 73

**6 Conclusion** **75**

6.1 Future Work . . . . . 76

6.1.1 Integration of Arithmetic Techniques . . . . . 76

6.1.2 Faults in other FPGA Components . . . . . 77

6.2 Closing Remarks . . . . . 77

# List of Tables


2.1 An example of a parity check matrix. . . . . 25

3.1 Truth table and CNF clauses for a 3-input look-up table . . . . . 33

3.2 Truth table and CNF clauses for a 3-input look-up table with SEUs . 35

3.3 Truth table and CNF clauses for $t_3$, which indicates if LUT three has an SEU . . . . . 37

3.4 CNF for a flip-flop with synchronous clear, synchronous set, synchronous data, enable, and data input signals. . . . . 38

3.5 QBF run times for a simple adder of varying size . . . . . 44

4.1 An example $H$ matrix and common fault set . . . . . 64

5.1 The area results for benchmark circuits after logic optimization and LUT mapping. . . . . 69

5.2 Results in terms of factored-form literals, and expressed as a percentage of the results in (Touba and McCluskey, 1997) . . . . . 71

5.3 Results of proposed procedure for an 8-bit AVR microcontroller. . . . 73

# List of Figures


2.1 Circuit diagram of a Cyclone II logic element taken from (Altera Corporation, 2007) . . . . . 13

2.2 Truth table and circuit diagram of a 3-input look-up table . . . . . 14

2.3 A typical FPGA tool flow . . . . . 15

2.4 An example of a Boolean network. . . . . 16

2.5 A diagram showing the difference between FS, SFS, TSC, and PFS networks (Smith and Metze, 1978) . . . . . 18

2.6 Circuit diagram of a design with concurrent error detection . . . . . 21

3.1 Circuit diagram of a flip-flop. . . . . 38

3.2 Diagram of the logic required to compute parity check bit $i$ . . . . . 41

3.3 A block diagram illustrating the structure of the CNF formula . . . . 42

3.4 Diagram showing logic shared between two outputs. . . . . 49

3.5 An example LUT network. . . . . 50

4.1 `PARITY_CHECK`: The algorithm used for selecting a parity check matrix. 66

# Chapter 1

# Introduction and Problem Statement

## 1.1 Safety-Critical Systems

A safety-critical system is a system whose failure could produce unacceptable effects, including loss of life or catastrophic damage to property or the environment (Knight, 2002). There are many modern applications where embedded computer systems are relied on to perform safety-critical tasks. In the automotive industry, numerous safety critical systems use computers for electronic control, including anti-lock braking systems and traction control (Baleani *et al.*, 2003). In the medical industry, insulin pumps and pacemakers are both examples of safety-critical systems that rely on computers for electronic control (Knight, 2002). Safety critical applications also exist in the military and aerospace industries. In nuclear power plants, computer systems are used to perform instrumentation and control tasks. In particular, nuclear power

plants have safety critical shutdown systems that are used to detect dangerous behaviour in the reactor and initiate a shut down procedure. Digital systems in nuclear power plant shut down applications form the initial motivation for this research.

Safety-critical computer systems have traditionally been implemented using software on microprocessor-based platforms because they presented the only way to develop a low volume, custom, programmable design. Because of the safety-critical nature of the system, it must be verified that these designs meet rigorous safety requirements before deployment. For microprocessor-based systems, this verification process can be more time consuming and costly than the implementation process. Besides verifying the correctness of the control software itself, the correctness of the hardware platform and lower levels of the software stack need to be taken into account, including any libraries that are used, scheduler, and operating system kernel. For safety-critical systems that are hard real-time systems, time-sensitive tasks must be scheduled such that the system meets stringent timing requirements. The design process, including specification, implementation, and verification can be extremely difficult, time-consuming, and costly (Wassyng *et al.*, 2005).

## 1.2 Field-Programmable Gate Arrays

Field programmable gate arrays (FPGAs) are a programmable technology that can implement a custom digital hardware design using primarily reconfigurable logic resources called look-up tables (LUTs) and flip-flops. The associated design flow looks similar to that which is used in an ASIC development process in that the design is implemented using a hardware description language (HDL) from which a netlist can be generated. The design is then mapped and routed to the logic resources on

the FPGA. The FPGA is programmed to implement the target hardware design by writing the configuration data to SRAM memory. SRAM cells are used to define the logic functions that are implemented in the LUTs of the FPGA and to implement the routing through interconnect switches that is required to connect the LUTs and flip-flops in the manner defined by the design netlist.

Since the initial patents for FPGAs were issued in the 1980s, they have emerged as a platform that can be used to implement safety-critical systems and an alternative to the conventional microprocessor-based designs. Although the design and verification time for hardware is generally much longer than for software designs, FPGAs can still offer some advantages over microprocessors for safety critical systems. For custom hardware designs which are implemented on FPGAs, functional verification can be more effective because the resulting system tends to be less complex than the microprocessor-based alternative. The FPGA-based system consists of only the design logic and the FPGA platform, while a microprocessor-based system would require verification of the entire software stack, as well as the application software and hardware platform. For hard real-time systems, a system can be designed and implemented on an FPGA that uses separate hardware state machines that operate concurrently to guarantee the timing for real-time tasks. This avoids the scheduling problems that can arise from faking concurrency by having multiple tasks running on the same hardware platform, and results in a system that is much easier to verify.

Another reason to use FPGAs in safety-critical systems is the design diversity it offers when paired with a microprocessor-based system (Avizienis and Kelly, 1984). Since the design process and tool flow for an FPGA-based system is most related to

hardware design, it is unlikely that an FPGA-based system and a microprocessor-based system would suffer from the same design errors. By designing a system that uses both, it is extremely unlikely for both systems to exhibit incorrect behaviour at the same time due to incorrect implementation of the system requirements. For applications where the safety requirement is more important than cost, such as the nuclear industry, this may be an attractive quality. Another case for using FPGAs in the nuclear industry is there ability to replace legacy microprocessor-based systems. In some cases, software has been designed and verified for a microprocessor that is no longer available. FPGAs have been proposed as a means of replacing obsolete components in safety-critical systems (Guzman-Miranda *et al.*, 2011). By emulating the legacy microprocessor on an FPGA, all the design and verification work for new software can be avoided by not having to switch to a new platform, although the FPGA-based emulator does need to be verified.

## 1.3 Single Event Upsets

For safety-critical systems, it is necessary to verify that the design not only meets the functional requirements, but also that it will continue to function correctly in the presence of environmental effects. A soft error or single event upset (SEU) is a change of state of any memory bit due to radiation in the environment. The result is a "bit flip" in flip-flops or memory blocks in a circuit, resulting in the effected bit being replaced by its complement. Single event upsets do not result in permanent damage to the circuit, but instead cause the circuit state or memory contents to be corrupted. For safety critical applications, it is necessary to account for the possibility of these effects and the problems they might cause to the system.

Modern FPGAs are configured through SRAM-based memory that has been shown to be susceptible to single event upsets (SEUs) due to natural radiation in the earth's atmosphere at aircraft altitudes and at ground level (Taber and Normand, 1993; Normand, 1996). Since the contents of the SRAM defines the hardware design to be implemented by the FPGA, SEUs in the configuration memory result in a change to the circuit functionality that persists until the FPGA is re-programmed. For LUTs, the logic function that is implemented by the LUT is changed, thus changing the functional behaviour of the design. For safety-critical systems, we assume that any incorrect design behaviour is unacceptable and may have serious consequences, so the system should be designed in such a way to mitigate these effects.

## 1.4 Problem Statement

In this chapter, the motivation has been established for examining the effects of single event upsets on FPGA-based safety-critical systems. The problem of mitigating against all kinds of faults in all memory elements of an FPGA is very broad and beyond the scope of this thesis. Instead, the focus of this research has been narrowed to address the problem of designing FPGA-based circuits that are able to detect the presence of faults in LUT configuration memory.

### 1.4.1 Fault Model

Despite the stringent safety requirements, it is impractical to design safety-critical systems such that system failure is impossible. The goal is not to design a system that cannot fail, but rather to limit the probability of failure to an acceptable value.

For soft errors in FPGAs, it is impossible to mitigate against any number of soft errors in any location. Instead, we identify a class of errors that are the most likely to occur and design our approach to detect all errors in the specified fault model. For the purposes of this thesis, the fault model is restricted to include single bit flips in LUT configuration memory.

### 1.4.2 Robustness Requirement

Although some safety-critical applications may have some inherent tolerance for faults, we would like a generalized approach that can meet the strictest safety requirements of any application. In developing an approach that can be used across many safety-critical applications, we assume that any externally observable faulty system behaviour is unacceptable. The approach developed through this thesis should be guaranteed to detect the first observable erroneous behaviour due to faults in the specified model. The resulting FPGA-based design should indicate that a fault has occurred within one clock cycle of the fault causing circuit outputs to be incorrect.

### 1.4.3 Cost Requirement

Although safety is the primary requirement for safety-critical systems, cost minimization forms a secondary objective. For FPGA-based systems, the logic resources required by the design should be minimized in order to minimize the cost of the FPGA platform itself. It has been widely acknowledged that circuit duplication is an easy means of identifying faulty behaviour. A successful approach should show an improvement over duplication in terms of the logic resources required by the design.

Cost minimization should be focussed not only around the cost of the FPGA platform itself, but also the non-recurring engineering cost of designing the circuit itself. The proposed technique should minimize the design time required for implementation of the circuit with fault detection. Ideally, the process should be automated and easily integrated with a typical FPGA design flow.

## 1.5 Contributions and Thesis Organization

This thesis will present a formal approach to designing FPGA-based systems that are able to detect the presence of faults in FPGA look-up tables. The circuit with a single SEU will be represented by a Boolean formula in conjunctive normal form (CNF). It will be shown that a quantified boolean formula (QBF) solver can be used to find a parity check matrix that can be used as the basis for concurrent error detection logic. The QBF-based approach will be shown to be limited for circuits of a realistic size.

An approach will then be discussed that involves using a satisfiability (SAT) solver to determine precisely how a LUT fault can affect circuit outputs. This information is sufficient to design a parity check matrix that guarantees detection of all single LUT faults. An algorithm will be presented to generate a parity check matrix that aims to minimize the logic resources required to implement the redundant logic. Run time and logic resource results for some benchmark circuits and a small microprocessor will be presented.

The rest of this thesis is organized as follows. Chapter 2 will provide some background on SAT and QBF solvers and FPGA architecture, followed by a survey of previous work on concurrent error detection. Chapter 3 will present a formal approach to error analysis of FPGA-based circuits subject to faults in the specified

model. Generation of a boolean formula in conjunctive normal form (CNF) will be discussed, as well as the scalability of techniques using SAT and QBF solvers. Chapter 4 will show how the results for Chapter 3 can be used to design a checking circuit based on parity check codes. Results will be presented and discussed in Chapter 5. Finally, Chapter 6 concludes the thesis and suggests some directions for future work.

# Chapter 2

# Background and Prior Work

In this chapter we will discuss some background material of relevance to fault mitigation in SRAM-based FPGAs. The chapter will start with an introduction to Boolean satisfiability, including the necessary background on conjunctive normal form (CNF), and QBF and SAT solvers. An overview of FPGA architecture with emphasis on the architectural details that are relevant to the techniques proposed in this thesis will then be given in Section 2.2. Section 2.3 will follow with an overview of prior work on fault tolerance of FPGA-based systems. After that, we will focus more specifically on concurrent error detection (CED), starting with some definitions related to self-checking circuits (Section 2.4). Section 2.5 will give an overview of prior work on circuits with CED. The chapter will conclude with a summary of these techniques and a justification of their unsuitability.

## 2.1 Boolean Satisfiability

Boolean satisfiability (SAT) is the problem of determining if a Boolean formula is satisfiable (Garey and Johnson, 1979). A Boolean formula is satisfiable if there exists an assignment to its variables which causes the formula to evaluate to true. The Boolean formula is generally specified in *conjunctive normal form* (CNF). CNF is a structure for Boolean formulas where the formula consists of a conjunction of clauses. A satisfying variable assignment must satisfy each of the clauses (each clause must evaluate to true). Each clause is composed of a disjunction of literals. For a clause to be satisfied, it must contain at least one literal which evaluates to true. Each literal is simply a variable in its negated or natural form. A literal evaluates to true if the variable is assigned to be true and it is not negated, or if the variable is assigned to be false and it is negated.

Consider the short Boolean formula in CNF form $(x \vee y) \wedge (\neg x \vee z)$. It contains two clauses, $(x \vee y)$ and $(\neg x \vee z)$, and both of these must evaluate to true for the formula to be true. The variable assignment $(x, y, z) = (0, 0, 1)$ would cause the first clause to evaluate to false, so the CNF formula would evaluate to false. Now consider the variable assignment $(x, y, z) = (0, 1, 0)$. The assignment $y = 1$ would cause the first clause to evaluate to true, and $x = 0$ would cause the second clause to evaluate to true, so the CNF formula would evaluate to true. This Boolean formula is satisfiable since there exists an assignment to its variables that causes it to evaluate to true, and $(x, y, z) = (0, 1, 0)$ is one such assignment.

### 2.1.1 SAT Solvers

A SAT solver is a software tool that is used to determine if a SAT instance in conjunctive normal form is satisfiable. The solver will either return 'UNSAT' (if the Boolean formula is unsatisfiable), or 'SAT' (if it is satisfiable). If the formula is satisfiable, the solver will generally return a variable assignment that causes the Boolean formula to evaluate to true.

An all-solutions SAT solver is one which is able to return the set of all satisfying assignments to some subset of the variables in the CNF (the "important" variables) (Grumberg *et al.*, 2004). One procedure for doing this is proposed in (Kang and Park, 2003). In this procedure, a SAT solver is used to find a single variable assignment for all the variables in the SAT instance. A clause is then added to the instance which blocks the solver from finding the same assignment to the important variables in the next run of the solver. This is done iteratively until the SAT solver returns 'UNSAT'.

The Boolean SAT problem is known to be NP-Complete, so there is no known efficient algorithm for solving all SAT instances, and the worst case running time for all known algorithms is exponential (Garey and Johnson, 1979). SAT solvers have been a subject of significant research, and solvers are able to solve many real-world instances efficiently (Balint *et al.*, 2012).

### 2.1.2 QBF Solvers

A Quantified Boolean Formula (QBF) is a generalization of SAT where variables are quantified using universal and existential quantifiers. The Boolean formula in CNF is preceded by a set of variable quantifiers that indicate universal quantification ($\forall$) or existential quantification ($\exists$). For example, the formula $\exists x \forall y \exists z (x \vee y) \wedge (\neg x \vee z)$

would say that there exists an $x$ such that for all $y$ there exists a $z$ such that the formula evaluates to true. Boolean SAT instances can be thought of as a special case of QBF instances where the prefix contains all variables with an existential quantifier. The QBF problem is known to be PSPACE-Complete (Garey and Johnson, 1979).

## 2.2 FPGAs

Before examining fault tolerant FPGA-based systems, it is necessary to understand some of the basics of FPGA architecture. Although we wish to establish techniques that are applicable to all FPGA families, Altera Cyclone II FPGAs and the associated Altera Quartus II tool flow form a representative implementation and are used as an example in this thesis.

### 2.2.1 Architecture Overview

Cyclone II devices contain a two-dimensional row-column based array of Logic Array Blocks (LABs) that are used to implement custom logic (Altera Corporation, 2007). A programmable interconnect is used to make connections between LABs across the FPGA. The LABs themselves are made up of sixteen smaller logic units, called Logic Elements (LEs). The circuit diagram for a Cyclone II logic element is included in Fig. 2.1. Each LE contains the fundamental building blocks needed to implement a digital circuit, namely, a look-up table (LUT) for implementing combinational logic, and a flip-flop, as well as connections to local (LAB wide) and global (FPGA wide) programmable interconnect. In addition to the LEs, Cyclone II FPGAs contain some

specialized hardware that can interface to the general logic resources, including multiplier blocks and embedded block RAMs.

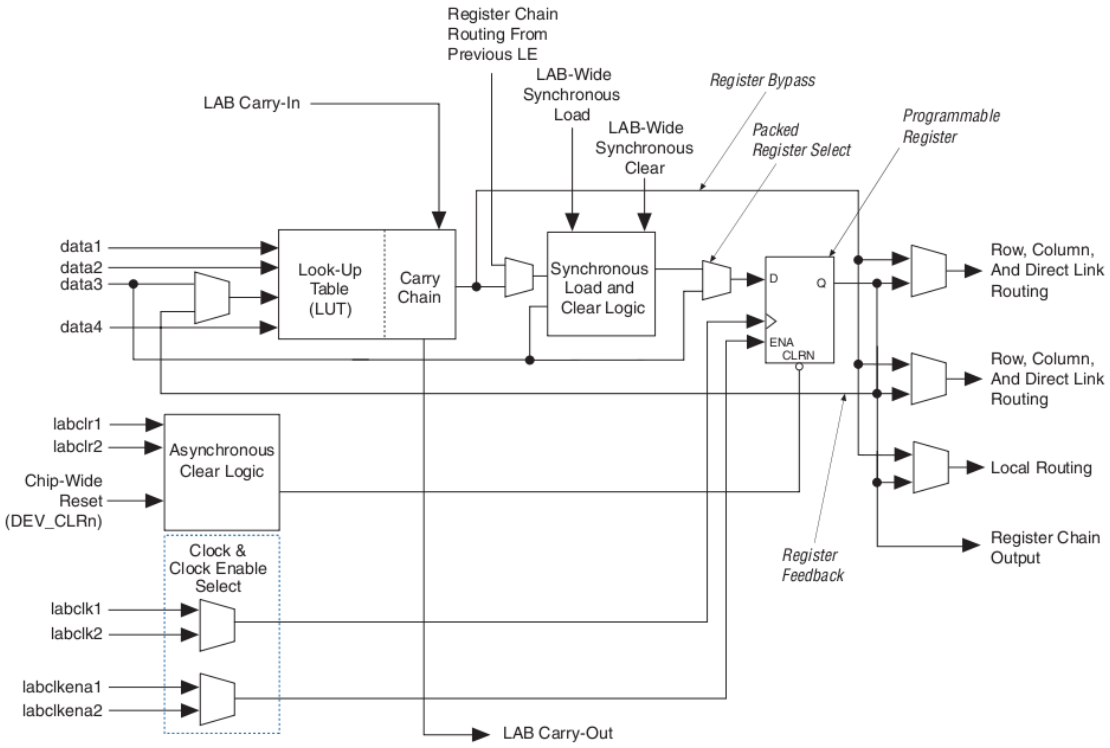


Figure 2.1: Circuit diagram of a Cyclone II logic element taken from (Altera Corporation, 2007)

### 2.2.2 Look-up Tables

Look-up tables (LUTs) are the fundamental building block that FPGAs use to implement combinational logic functions. An N-input LUT can used to implement any combinational function of N inputs. For Cyclone II FPGAs, 4-input LUTs are used, however 6-input LUTs are common on more advanced devices. The truth table and circuit diagram for a LUT can be found in Fig. 2.2. The truth table of the logic

function to be implemented is stored in SRAM, and multiplexers are used to select the row of the truth table that corresponds to the value driven on the LUT inputs.

| $w$ | $x$ | $y$ | $z$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

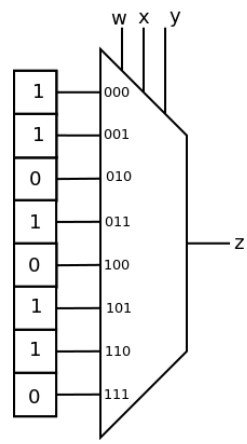


Figure 2.2: Truth table and circuit diagram of a 3-input look-up table

### 2.2.3 Tool Flow

The tool flow for FPGA-based designs involves multiple stages that take a hardware description language (HDL) implementation, to programming the physical SRAM cells on the FPGA. A typical FPGA tool flow is presented in Fig. 2.3. The synthesis stage involves translation of the HDL description into an optimized network of flip-flops and combinational logic equations. During the technology mapping stage, combinational logic is mapped onto LUTs, and the result is a network of flip-flops and LUTs and their configuration data. For the Quartus II tool flow used in this thesis, the result of the technology mapping stage can be output in a Verilog Quartus Mapping (VQM) file, which is a Verilog file showing the interconnection of LUT and flip-flop instances. The place and route stage takes care of allocating specific LUTs on the FPGA to be used to implement the design. During FPGA configuration, the bit stream is produced that is used during programming.

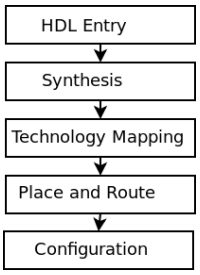


Figure 2.3: A typical FPGA tool flow

## 2.3 Fault Tolerance in FPGAs

Prior work on fault tolerant FPGA-based systems includes a variety of approaches, and many of them do not meet the requirements set out in Chapter 1 of this thesis. The simplest approach to fault tolerance involves simply using multiple copies of the circuit. Triple-modular redundancy (TMR) involves creating three copies of the same circuit (Lyons and Vanderkulk, 1962). A voter circuit is added that compares the outputs and takes the two that agree to be correct. Assuming there is a fault in only one copy of the circuit, the circuit can continue normal operation. Circuit duplication is a similar technique that involves create two copies of the circuit. The circuit outputs are compared, and a fault is detected if they are different. With duplication, no fault recovery or correction is possible because it is impossible to tell which output is the correct one. Both TMR and circuit duplication represent general techniques which can be applied toward FPGA-based designs.

The techniques proposed in (Hu *et al.*, 2008), (Feng *et al.*, 2009), (Feng *et al.*, 2011), and (Jing *et al.*, 2011) all seek to mitigate against the effects of SEUs in LUT and interconnect configuration data in FPGAs by performing modifications to the way the circuit is mapped to LUTs without causing an increase in the resource required. Although they are able to decrease the percentage of faults which cause the circuit

to exhibit incorrect behaviour, they are not able to guarantee that no faults will cause incorrect circuit outputs without being detected. In general, techniques that seek to lower the percentage of faults that go undetected compose a large portion of techniques for developing fault tolerant FPGA-based systems.

Fault tolerant architectures for FPGA logic resources are discussed in (Reddy *et al.*, 2005). Two architectures are proposed, the first is able to detect faults during normal operation, while the second is able to detect faults in three clock cycles while the circuit pauses its normal operation. Since this technique involves designing an FPGA with an entirely new architecture, it does not meet the specified requirements.

## 2.4 The Totally Self-Checking Goal

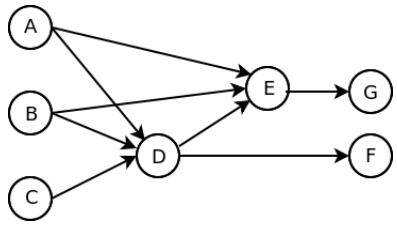


Figure 2.4: An example of a Boolean network.

At this time, some basic definitions of relevance to error detection in digital circuits are presented. A *Boolean network* is a way of representing a multi-level digital circuit as a directed acyclic graph (DAG) (Brayton *et al.*, 1990). Each node in the network has a set of inputs, a single output, and an associated Boolean function by which the output can be computed from the set of inputs. An edge in the network from node i to node j indicates that the output of node i is an input to node j. Primary inputs and outputs to the circuit are represented by special nodes which compute no

logic function. Consider the Boolean network presented in Fig. 2.4. Nodes A, B, and C would be primary inputs to the network, and nodes F and G would be primary outputs. Node D would compute a Boolean function of A, B, and C, while node E would compute a Boolean function of A, B, and D. In an FPGA, where logic functions are implemented using LUTs, the design can be represented with a Boolean network where each N-input LUT is represented by a node in the network with N inputs. In Fig. 2.4, nodes D and E could represent 3-input LUTs.

Consider a digital circuit that is represented by a Boolean network. For a circuit with N primary inputs, there are $2^N$ possible input vectors to the circuit. We call this set the *input space*. We likewise define the *output space* to be the set of all possible output vectors. During fault free operation, only a subset of the input space called the *input code space* is applied to the network. The outputs of the circuit during fault free operations form the *output code space*. Vectors that are in the code space are referred to as *code words*, and vectors that are not in the code space are referred to as *noncode words* (Smith and Metze, 1978). Any time the network produces a noncode word output, this indicates the presence of a fault in the network.

When designing a circuit that is able to detect the presence of faults, the goal is that the corresponding Boolean network produces a noncode word output the first time the output is erroneous due to a fault. This is referred to in literature as the *Totally self-checking goal*, and leads to five definitions about Boolean networks that will be discussed in the remainder of this section.

A Boolean network is considered to be *fault secure* (FS) with respect to a particular set of faults, if, for any fault in the set, the output of the network is either correct, or it is a noncode word. A Boolean network is considered to be *self-testing* if, for every

fault in the fault set, there exists some code word input such that the output is a noncode word. A circuit is considered to be *totally self-checking* (TSC) if it is both *fault secure* and *self-testing* (Smith and Metze, 1978).

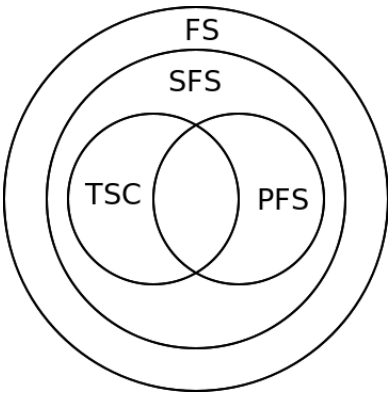


Figure 2.5: A diagram showing the difference between FS, SFS, TSC, and PFS networks (Smith and Metze, 1978)

The effectiveness of *totally self-checking* networks for error detection is a result of some assumptions about the behaviour of faults. We assume that faults occur one at a time from a specified fault class. We further assume that there is a lower bound on the amount of time that passes between consecutive faults. We assume that, during normal operation, there is an upper bound on the time required to apply all code word inputs to the network, and this upper bound is less than the minimum time between faults. In this way, we are assuming that all code word inputs are applied to the network betweeen consecutive faults due to the behaviour of faults and the behaviour of the network.

The reason for the fault secure requirement for totally self-checking networks should be obvious; it is our goal to "catch" the first occurrence of an incorrect output, and the fault secure property guarantees that any incorrect output is also a noncode word. The self-testing requirement is less obvious. If the network were not

self-testing, it might be possible for a fault to not cause an incorrect output under any code word input (and hence it would never be caught). If a sufficient amount of time passes, a second fault might occur, and the fault secure property would not guarantee detection of the first incorrect output with the presence of two faults. The self-testing property guarantees that all faults are "exposed" by some code word input. This way, it is guaranteed that any fault will be caught before the next one occurs (since we have assumed that all code word inputs are applied before the next fault happens) (Smith and Metze, 1978).

Two other types of networks that meet the totally self-checking goal are described in (Smith and Metze, 1978). *Strongly fault secure* (SFS) networks are a subset of fault secure networks that meet the totally self-checking goal even though they are not also self-testing. Recall that TSC networks are required to be self-testing so that multiple faults cannot combine to cause incorrect outputs that go undetected. SFS networks are networks where this is not possible even though the network is not self-testing. For SFS networks, it is proven that the first set of faults that causes an incorrect output for some code word input causes either correct outputs or non code word outputs for all code word inputs. In this way, no sequence of faults can cause incorrect outputs that are undetected.

*Path fault secure* (PFS) networks are networks in which the FS property is guaranteed by the structure of the network (Smith and Metze, 1978). In a PFS network, it is assumed that any fault can propagate down any structural path from the fault location to the network outputs. A network is PFS if and only if no fault in the specified class can produce an incorrect code word output, even if the fault propagates down *any* structural path from its location to the network output. It is shown

in (Smith and Metze, 1978) that any network that is PFS is also SFS and therefore meets the totally self-checking goal. A diagram illustrating the relationship between FS, SFS, PFS, and TSC networks is given in Fig. 2.5.

## 2.5 Concurrent Error Detection

Concurrent error detection (CED) is a technique that involves using redundancy to detect faults in digital hardware designs. Although we are considering fault detection in hardware that is implemented on an FPGA, the majority of previous work has been related to the detection of faults in integrated circuits themselves. In general, these approaches consider stuck-at faults in circuit nets as the fault model. A stuck-at fault is fault in which a wire in the design no longer exhibits the correct logical behaviour and becomes stuck, driving either one or zero (Wadsack, 1978). These techniques are considered because of their relevance to concurrent error detection, even though they are not necessarily directly transferable to FPGA-based designs.

The most common technique for implementing designs with concurrent error detection involves synthesizing some extra hardware that independently predicts properties about the circuit output (Mitra and McCluskey, 2000). A circuit diagram is included in Fig. 2.6. For the purposes of this thesis, we will refer to the original design logic as the *function logic*. The extra logic is generally used to compute a type of error detecting code that can be checked against the output to determine if there is a fault. We refer to this redundant logic as the *check symbol predict logic* or simply the redundant logic. If the outputs to the function logic and the check symbol predict logic are considered to be the primary outputs to the related Boolean network, any output vector where the check symbols are consistent with the function logic outputs

would be considered to be the output code space of the network. If the check symbols are inconsistent with the function outputs, the output vector would be considered a noncode word and would indicate a fault in the circuit. The idea would be to select a check symbol that forms a noncode word any time an erroneous output is caused by a fault in the considered fault model, thus forming a *fault secure* circuit (Mitra and McCluskey, 2000). The type of check symbol to use has been a topic of considerable research and will be discussed in Sections 2.5.1, 2.5.2, and 2.5.3.

Typically a checker is added to determine if the output vector forms a code word or a noncode word. The full circuit with a checker is what is presented in Fig. 2.6.

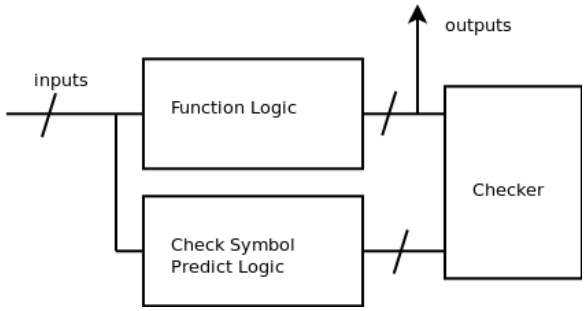


Figure 2.6: Circuit diagram of a design with concurrent error detection

### 2.5.1 Berger Codes

Berger codes are a type of systematic code that can be used for detection of unidirectional errors (Berger, 1961). Unidirectional errors occur in applications where faults can only cause zeros to flip to ones, or only cause ones to flip to zeros. The Berger code for N information bits is computed by counting the number of zeros that appear in the data. The Berger code simply stores that number in binary. For N information bits, the maximum value of the Berger code is N (when all information bits are zero),

so the Berger code requires an additional $\log_2(N+1)$ additional bits. Berger codes have been shown to be optimal codes for detecting unidirectional bit errors (Freiman, 1962).

Berger codes have been proposed as a method of designing circuits with concurrent error detection in (Lo *et al.*, 1989) and (Fuchs *et al.*, 1987). The idea would be to have the check symbol predict logic compute the number of zeros that should be found in the output of the function logic. The checker would compare the number of zeros that actually occur in the outputs of the function logic and compare the result to the Berger code. Note that a single stuck-at fault in the function logic could cause multiple outputs to be incorrect if the effected net is included in the input cone of multiple outputs. Since Berger codes only detect unidirectional errors, the function logic would have to be designed such that a single stuck-at fault can only cause unidirectional errors in its outputs. A solution has been proposed in (Jha and Wang, 1991) that involves synthesizing the function logic such that inverters only occur at the inputs. The remainder of the circuit is inverter-free, and a stuck-at fault can only cause unidirectional errors in the outputs to the function logic.

Although Berger codes are optimal for detecting all unidirectional bit errors in data, that does not imply that Berger codes are the best choice as a basis for designing hardware systems with concurrent error detection. With concurrent error detection, the goal is to minimize the resources required to implement the check symbol predict logic. Although minimizing the number of check bits may help with this goal, there is no guarantee that a code requires less logic resources to compute, simply because it uses fewer bits. In addition, the structural constraints that are imposed on the function logic to make it inverter free will likely cause an increase in the resources

require by the function logic when compared to an unconstrained implementation.

### 2.5.2 Bose-Lin Codes

Bose-Lin codes are a type of systematic code that is related to Berger codes, and like Berger codes, they are used to detect unidirectional bit errors. They are fundamentally similar in that they also perform a summation of the number of zeros in the information bits and use that information in the check bits. While Berger codes include the full number of zeros and thus require $\log_2(N+1)$ check bits, Bose-Lin codes cut down on the number of check bits by storing only some least significant bits of this summation. For Bose-Lin codes on $2 \leq t \leq 3$ bits, the check bits simply store the number of zeros in the information bits modulo t (Bose and Lin, 1985). A Bose-Lin code is represented on fewer bits than a Berger code, and the cost for that is decreased capacity to detect errors. A t-bit Bose-Lin code is guaranteed to catch unidirectional bit flips as long as the number of effected bits is less than $2^t$. Bose-Lin codes on $t \geq 4$ bits are beyond the scope of this thesis, and they are explained further in (Bose and Lin, 1985).

Circuits with concurrent error detection based on Bose-Lin codes have been proposed in (Gorshe and Bose, 1996) and (Das and Touba, 1998). In (Gorshe and Bose, 1996), the feasibility of using Bose-Lin codes to detect errors in a simple ALU is examined. It is shown that the proposed architecture is fault secure with respect to single stuck-at faults due to the nature of the arithmetic and logical operations of the ALU. This is despite the reduction in the number of check bits as compared to using Berger codes. Bose-Lin codes using $2 \leq t \leq 3$ bits were shown to offer a savings in logic resources when compared to using Berger codes, but Bose-Lin codes of $t \geq 4$

bits were shown to offer insignificant savings due to the increased complexity of the hardware required to compute these codes.

In (Das and Touba, 1998), Bose-Lin codes were proposed as a basis for designing concurrent error detection logic for circuits implementing arbitrary function logic. Bose-Lin codes using $t = 2$ bits were examined. In general, it is possible for a single stuck-at fault to result in a error in any circuit output that has the affected net in its input cone. Because it is an inherent limitation of Bose-Lin codes, the check symbol on $t = 2$ bits is only capable of detecting errors in $2^t = 4$ outputs. To prove that the circuit is fault secure, it is necessary to prove that no stuck at fault can cause four or more outputs to be incorrect. The proposed procedure involves a multi-level synthesis procedure for the function logic that prevents a net from appearing in the input cone of four or more outputs. This is in addition to the inverter-free constraint as described in the previous section. The results show that CED schemes based on Bose-Lin codes require fewer logic resources than those based on Berger codes or using duplication.

### 2.5.3 Parity Check Codes

Parity check codes are another type of systematic code that can be used for error detection. A parity bit is a single bit that represents the parity of a group of information bits. For odd parity, the parity bit is the modulo 2 sum of the information bits, or equivalently, the parity is one if the number of ones in the information bits is odd and zero if it is even. A parity check code is composed of a number of parity bits where each bit is the parity of a subset of the information bits. The design of the parity check code is represented by the parity check matrix $H$, and an example

is shown in Table. 2.1 (Pradhan, 1986). A one in row $i$ and column $j$ of the parity check matrix indicates that information bit $j$ is checked by parity bit $i$. For the example $H$ matrix in Table 2.1, parity bit 1 computes the parity of information bits 1 and 2, parity bit 2 computes the parity of information bit 2 and 4, and parity bit 3 computes the parity of information bit 5 (which is the same as duplicating bit 5). The group of information bits that is checked by a single parity bit will be referred to as a *parity group*. A parity bit is capable of detecting any single bit flip in the related information bits, but if two bit flips occur, the parity of the information will be unchanged and the fault will not be detected. In general, a parity bit will detect a fault in the information bits if they contain an odd number of bit flips. For parity check codes composed of multiple parity bits, a fault need only be detected by one of the parity bits.

| | | Information Bit | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 |
| Parity Bit | 1 | 1 | 1 | 0 | 0 | 0 |
| | 2 | 0 | 1 | 0 | 1 | 0 |
| | 3 | 0 | 0 | 0 | 0 | 1 |

Table 2.1: An example of a parity check matrix.

For concurrent error detection schemes based on parity check codes, each of the outputs of the check symbol predict logic would be the parity of a group of outputs of the function logic. For a parity check code based on the matrix presented in Table 2.1, output 2 of the check symbol predict logic would be the parity of outputs 2 and 4 of the function logic. To guarantee that the circuit is fault secure, a parity check code has to be selected such that any fault in the function logic that causes any output to be incorrect is detected by at least one of the parity bits (ie, it must cause an odd number of errors in the outputs checked by that parity bit).

A technique for designing concurrent error detection logic based on a parity check code is presented in (De *et al.*, 1994). The function outputs are divided into output groups, where each group is a disjoint subset of the function outputs and each output appears in exactly one output group. When logic optimization is done on the function logic, output groups are optimized separately so unlimited logic sharing is allowed within an output group, but outputs in different output groups cannot share logic at all. If any stuck-at fault causes multiple outputs to be incorrect, all the incorrect outputs will be in the same output group. The parity check matrix is selected such that no parity group contains more than one output from any output group. This guarantees that a parity group can only contain one incorrect output, and the associated parity bit will detect the error. In this way, the circuit can be shown to be fault secure. When compared to fault secure circuits using Berger codes, the parity check scheme is shown to use fewer logic resources.

Another technique for generating a fault secure circuit using a parity check code is proposed in (Touba and McCluskey, 1997). In this approach, a synthesis procedure called *structure-constrained logic optimization* (SCLO) is introduced. Using SCLO, the function logic can undergo logic optimization that honours structural constraints about which outputs can share logic resources. Parity groups form disjoint sets of function logic outputs. If two outputs are checked by the same parity bit, then they are not allowed to share logic resources, and a constraint in the synthesis procedure prevents this from happening. Since outputs in the same parity group are not allowed to share logic resources, it is guaranteed that a stuck-at fault cannot affect more than one output in a parity group, thus the first fault that causes an incorrect output will be detected (the fault secure property). A method for estimating the logic resources

required by different parity check codes is introduced that includes estimating the size of the check symbol predict logic, the size of the checker, and the amount of extra resources required due to added logic sharing constraints. An algorithm is presented to use this information to select an parity check matrix. This technique is shown to require fewer logic resource than the technique presented in (De *et al.*, 1994).

### 2.5.4 FPGA-based Approaches

The previously discussed approaches to designing circuits with concurrent error detection have all been targeted towards detection of stuck-at faults. For FPGA-based designs, we are primarily concerned with SEUs in FPGA configuration memory, so techniques that detect stuck-at faults in integrated circuits use a different fault model. The remainder of this section will discuss some techniques for concurrent error detection that are targeted towards FPGAs.

An approach is proposed in (Krasniewski, 2006) that involves using embedded memory blocks in FPGAs to implement finite state machines by storing all the state transitions and circuit outputs in the memory. The results are shown for circuits of less than seven inputs, sixteen outputs, and twenty states, so the techniques do not represent something that would be useful for circuits of more significant size.

Recently, Altera FPGAs are designed with dedicated error detection circuitry that computes a cyclic redundancy check (CRC) of the entire configuration memory with a latency in the order of tens of milliseconds (Altera Corporation, 2008). Although this would guarantee the eventual detection of any error in FPGA configuration bits, it does not ensure that the error is caught the first time a circuit output is incorrect (ie. the fault secure property). For TSC circuits, the self-testing requirement is intended

to guarantee that any fault is detected before another one can occur, so the CRC check could be used to fulfill this requirement if we assume that the time between faults is always greater than the time required to perform the CRC check.

An attempt is made in (Bolchini *et al.*, 2002) to take concurrent error detection methods using systematic codes and implement them as a way to detect faults in LUT configuration memory and stuck-at faults at LUT I/O in FPGAs. It is acknowledged that these techniques all rely on structural properties of the design to guarantee the detection of all faults, and that commercial FPGA mapping tools are not designed to maintain these properties during the LUT mapping process. The proposed technique performs this LUT mapping and then does an analysis of the fault coverage using simulation. When undetected faults are discovered, some local modifications to the LUT network are performed in an attempt to increase detection. The results show a significant increase in the number of LUTs needed to implement the design, while the technique is still unable to guarantee detection of all faults.

## 2.6 Summary of Prior Work

In general, there has not been extensive research published on fault tolerance techniques that are targeting specifically to FPGAs and meet the goals described in Chapter 1 of this thesis. The majority of fault tolerance techniques for FPGAs either fail to show detection/correction of all faults or achieve fault tolerance through modified FPGA architectures. The FPGA-based approach to concurrent error detection presented in (Krasniewski, 2006) has been shown to be ineffective for dealing with circuits of significant size. The CRC check that is available in many modern FPGAs is unable to produce a circuit that is fault secure, but it could be used to satisfy

the self-testing requirement of TSC circuits (Altera Corporation, 2008). We focus on satisfying the fault secure requirement in the remainder of this thesis.

We now review the best approach for designing concurrent error detection logic using systematic codes. It is shown that CED logic based on parity check codes requires fewer logic resources than techniques which use Berger codes (De *et al.*, 1994; Touba and McCluskey, 1997). A discussion of different error detection schemes is presented in (Mitra and McCluskey, 2000), and it is indicated that CED logic based on parity check codes requires fewer logic resources than techniques which use Berger codes or Bose-Lin codes.

Concurrent error detection schemes that use systematic codes to develop TSC circuits have been identified as error detection methods which meet our requirements, but these approaches have not been successfully implemented on FPGAs. The approach described in (Bolchini *et al.*, 2002) fails to guarantee full fault coverage when implementing these techniques on FPGAs, and indicates that current technology mapping tools are the reason for this. The problem of designing a suitable LUT mapping tool for maintaining structural constraints appears to be non trivial and could be a direction for research itself. For safety-critical FPGA-based systems, the designer must have a high degree of confidence that the software tool flow is correct and will not introduce errors into the design. If such a LUT mapping tool could be designed, the tool itself would have to be verified. A solution that allows the function logic to go through an established, commercially available tool flow would be preferable to one that involves a custom tool flow. Such a technique will be presented in Chapters 3 and 4 of this thesis.

# Chapter 3

# Formal Approach to CED

## 3.1 Introduction

In Chapter 2, it was identified that concurrent error detection techniques using systematic codes satisfy the requirements laid out in the first chapter of this thesis. When concurrent error detection logic is used to produce totally self-checking circuits, it can be shown that such circuits guarantee the detection of errors on the first occurrence of an incorrect circuit output due to stuck-at faults under some assumptions. These techniques all rely on structural constraints on logic optimization to guarantee the fault secure property, and they have not been adapted to work with FPGA-based implementations. Among CED techniques, those that use redundant logic based on parity check codes have been shown to use the fewest logic resources on average, so we use parity check codes as the basis for the proposed technique.

For the technique used in (Touba and McCluskey, 1997), the function logic is optimized under constraints that assume that any two outputs that share logic could both be incorrect due to a fault in that shared logic, and on this basis, a parity check

code is designed. Consider a net that is in the input cone for N different outputs. Under these assumptions about how faults can affect circuit outputs, these outputs could be incorrect in any combination, resulting in $2^N$ possible combinations. In reality, faults can be masked inside the circuit under some inputs, and this number of combinations may not be possible. In fact for some faults, there may not be an input vector that causes any output to be incorrect (so the circuit is not self-testing), but the PFS property guarantees that the circuit is SFS, and that the totally self-checking goal is achieved.

For FPGA-based designs, a fault in LUT configuration memory can only cause errors in circuit outputs if LUT inputs select that bit to drive the output. Some LUT configuration bits may not cause any circuit outputs to be incorrect for any circuit inputs, thus violating the self-testing property that is required for TSC circuits. For FPGA-based designs, we are unconcerned about designing our circuit to satisfy the self-testing property since the CRC check that is built into many FPGAs can be used to satisfy this property (Altera Corporation, 2008). The proposed technique only needs to guarantee that the circuit is fault secure. To this end, fault masking is to our advantage since a masked fault does not affect circuit outputs, and we do not need to design a parity check matrix that can detect it if we can identify precisely how circuit outputs can be affected by faults.

This chapter will present a formal approach to examining how LUT faults affect the outputs of FPGA-based hardware designs. Instead of selecting a parity check code and then optimizing the function logic with constraints to suit the selected code, we start by performing unconstrained logic optimization and technology mapping of the function logic. The result is a network of LUTs and flip-flops that implements the

function logic. A methodology will be presented for generating CNF clauses that represent the LUTs and flip-flops in the design. A technique for using a QBF solver to find a parity check matrix which results in a fault secure circuit will then be described. Although this technique is complete and has been used successfully on small circuits, it will shown to be impractical for designs of realistic size. Last, SAT-based approaches will be presented, including a method for verifying that a parity check matrix satisfies the fault secure property and a method for extracting sets of incorrect outputs which are uncaught by a parity check matrix and violate the fault secure property. An algorithm that can be used to generate a parity check matrix that will detect all errors is presented in Chapter 4.

## 3.2 CNF Formula of a Circuit

For our formal approach to developing concurrent error detection logic based on parity check codes, the first step is generating a CNF formula that can be used to represent the design. At this stage of the development process, the design is not represented by a Boolean network (which is an interconnection of signals and their logic equations), but instead by a network of FPGA logic elements (LUTs and flip-flops). The CNF formula uses a set of variables that represent signals in the design and constraints (in the form of CNF clauses) that are used to represent the design logic. When evaluating the CNF formula with a solver, a satisfying variable assignment gives the Boolean value driven on each signal in the design under some input vector.

| $w$ | $x$ | $y$ | $z$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

| | | | |
|---|---|---|---|
| $w$ | $x$ | $y$ | $z$ |
| $w$ | $x$ | $\neg y$ | $z$ |
| $w$ | $\neg x$ | $y$ | $\neg z$ |
| $w$ | $\neg x$ | $\neg y$ | $z$ |
| $\neg w$ | $x$ | $y$ | $\neg z$ |
| $\neg w$ | $x$ | $\neg y$ | $z$ |
| $\neg w$ | $\neg x$ | $y$ | $z$ |
| $\neg w$ | $\neg x$ | $\neg y$ | $\neg z$ |

Table 3.1: Truth table and CNF clauses for a 3-input look-up table

### 3.2.1 Look-up Tables with SEUs

For combinational logic functions, a clause is used to constrain the output for each row of the truth table. Consider the 3-input LUT and its truth table in Fig. 2.2. The resulting CNF clauses are shown next to the truth table in Table 3.1. Each clause contains each input variable, and the variable is negated if it is assigned a one in that row of the truth table and not negated if it is assigned a zero. The output variable also appears in each clause and is negated if it is assigned a zero in that row and not negated if it is assigned a one. Consider the second row of the truth table in Table 3.1. The resulting cause is $w \vee x \vee \neg y \vee z$. If the input variables $(w, x, y)$ have the values $(0, 0, 1)$ in a satisfying variable assignment, the first three literals in the clause all evaluate to zero, so the literal containing the $z$ (the output of the truth table) must be one in order to satisfy the clause. If $(w, x, y)$ take some value other than $(0, 0, 1)$ in a satisfying variable assignment, then one of the first three literals in the clause will evaluate to one, and the clause will be satisfied. In this way, the clause $w \vee x \vee \neg y \vee z$ means that a variable assignment where $(w, x, y) = (0, 0, 1)$ must also have $z = 1$, but if $(w, x, y) \neq (0, 0, 1)$, then no constraint is placed on $z$. In this way, the variables $(w, x, y, z)$ are forced to behave according to the truth table in a

satisfying variable assignment.

For an $N$ input LUT, this results in $N + 1$ variables. The number of clauses required is $2^N$ for each LUT.

This method for translating LUTs into the CNF formula results in constraints that cause the LUT output to compute the desired truth table. For analysis of the design with the presence of SEUs in the LUT configuration memory, we actually want to be able to add constraints such that there is a single fault in a LUT configuration bit. Four new variables are introduced, $r_0$-$r_2$ and t. The first three ($r_0$-$r_2$) indicate which LUT configuration bit has a fault. The fourth, t, indicates whether the LUT has a fault. The full truth table and CNF clauses for a LUT with these four addition inputs is given in Table 3.2 with rows of the truth table in bold indicating variable assignments that result in an SEU affecting the LUT output.

| $t$ | $r_2$ | $r_1$ | $r_0$ | $w$ | $x$ | $y$ | $z$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | $\neg t$ | $r_2$ | $r_1$ | $r_0$ | $w$ | $x$ | $y$ | $\neg z$ |
| 0 | x | x | x | 0 | 0 | 0 | 1 | $t$ | | | | $w$ | $x$ | $y$ | $z$ |
| x | 1 | x | x | 0 | 0 | 0 | 1 | | $\neg r_2$ | | | $w$ | $x$ | $y$ | $z$ |
| x | x | 1 | x | 0 | 0 | 0 | 1 | | | $\neg r_1$ | | $w$ | $x$ | $y$ | $z$ |
| x | x | x | 1 | 0 | 0 | 0 | 1 | | | | $\neg r_0$ | $w$ | $x$ | $y$ | $z$ |
| **1** | **0** | **0** | **1** | **0** | **0** | **1** | **0** | $\neg t$ | $r_2$ | $r_1$ | $\neg r_0$ | $w$ | $x$ | $\neg y$ | $\neg z$ |
| 0 | x | x | x | 0 | 0 | 1 | 1 | $t$ | | | | $w$ | $x$ | $\neg y$ | $z$ |
| x | 1 | x | x | 0 | 0 | 1 | 1 | | $\neg r_2$ | | | $w$ | $x$ | $\neg y$ | $z$ |
| x | x | 1 | x | 0 | 0 | 1 | 1 | | | $\neg r_1$ | | $w$ | $x$ | $\neg y$ | $z$ |
| x | x | x | 0 | 0 | 0 | 1 | 1 | | | | $r_0$ | $w$ | $x$ | $\neg y$ | $z$ |
| **1** | **0** | **1** | **0** | **0** | **1** | **0** | **1** | $\neg t$ | $r_2$ | $\neg r_1$ | $r_0$ | $w$ | $\neg x$ | $y$ | $z$ |
| 0 | x | x | x | 0 | 1 | 0 | 0 | $t$ | | | | $w$ | $\neg x$ | $y$ | $\neg z$ |
| x | 1 | x | x | 0 | 1 | 0 | 0 | | $\neg r_2$ | | | $w$ | $\neg x$ | $y$ | $\neg z$ |
| x | x | 0 | x | 0 | 1 | 0 | 0 | | | $r_1$ | | $w$ | $\neg x$ | $y$ | $\neg z$ |
| x | x | x | 1 | 0 | 1 | 0 | 0 | | | | $\neg r_0$ | $w$ | $\neg x$ | $y$ | $\neg z$ |
| **1** | **0** | **1** | **1** | **0** | **1** | **1** | **0** | $\neg t$ | $r_2$ | $\neg r_1$ | $\neg r_0$ | $w$ | $\neg x$ | $\neg y$ | $\neg z$ |
| 0 | x | x | x | 0 | 1 | 1 | 1 | $t$ | | | | $w$ | $\neg x$ | $\neg y$ | $z$ |
| x | 1 | x | x | 0 | 1 | 1 | 1 | | $\neg r_2$ | | | $w$ | $\neg x$ | $\neg y$ | $z$ |
| x | x | 0 | x | 0 | 1 | 1 | 1 | | | $r_1$ | | $w$ | $\neg x$ | $\neg y$ | $z$ |
| x | x | x | 0 | 0 | 1 | 1 | 1 | | | | $r_0$ | $w$ | $\neg x$ | $\neg y$ | $z$ |
| **1** | **1** | **0** | **0** | **1** | **0** | **0** | **1** | $\neg t$ | $\neg r_2$ | $r_1$ | $r_0$ | $\neg w$ | $x$ | $y$ | $z$ |
| 0 | x | x | x | 1 | 0 | 0 | 0 | $t$ | | | | $\neg w$ | $x$ | $y$ | $\neg z$ |
| x | 0 | x | x | 1 | 0 | 0 | 0 | | $r_2$ | | | $\neg w$ | $x$ | $y$ | $\neg z$ |
| x | x | 1 | x | 1 | 0 | 0 | 0 | | | $\neg r_1$ | | $\neg w$ | $x$ | $y$ | $\neg z$ |
| x | x | x | 1 | 1 | 0 | 0 | 0 | | | | $\neg r_0$ | $\neg w$ | $x$ | $y$ | $\neg z$ |
| **1** | **1** | **0** | **1** | **1** | **0** | **1** | **0** | $\neg t$ | $\neg r_2$ | $r_1$ | $\neg r_0$ | $\neg w$ | $x$ | $\neg y$ | $\neg z$ |
| 0 | x | x | x | 1 | 0 | 1 | 1 | $t$ | | | | $\neg w$ | $x$ | $\neg y$ | $z$ |
| x | 0 | x | x | 1 | 0 | 1 | 1 | | $r_2$ | | | $\neg w$ | $x$ | $\neg y$ | $z$ |
| x | x | 1 | x | 1 | 0 | 1 | 1 | | | $\neg r_1$ | | $\neg w$ | $x$ | $\neg y$ | $z$ |
| x | x | x | 0 | 1 | 0 | 1 | 1 | | | | $r_0$ | $\neg w$ | $x$ | $\neg y$ | $z$ |
| **1** | **1** | **1** | **0** | **1** | **1** | **0** | **0** | $\neg t$ | $\neg r_2$ | $\neg r_1$ | $r_0$ | $\neg w$ | $\neg x$ | $y$ | $\neg z$ |
| 0 | x | x | x | 1 | 1 | 0 | 1 | $t$ | | | | $\neg w$ | $\neg x$ | $y$ | $z$ |
| x | 0 | x | x | 1 | 1 | 0 | 1 | | $r_2$ | | | $\neg w$ | $\neg x$ | $y$ | $z$ |
| x | x | 0 | x | 1 | 1 | 0 | 1 | | | $r_1$ | | $\neg w$ | $\neg x$ | $y$ | $z$ |
| x | x | x | 1 | 1 | 1 | 0 | 1 | | | | $\neg r_0$ | $\neg w$ | $\neg x$ | $y$ | $z$ |
| **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | $\neg t$ | $\neg r_2$ | $\neg r_1$ | $\neg r_0$ | $\neg w$ | $\neg x$ | $\neg y$ | $z$ |
| 0 | x | x | x | 1 | 1 | 1 | 0 | $t$ | | | | $\neg w$ | $\neg x$ | $\neg y$ | $\neg z$ |
| x | 0 | x | x | 1 | 1 | 1 | 0 | | $r_2$ | | | $\neg w$ | $\neg x$ | $\neg y$ | $\neg z$ |
| x | x | 0 | x | 1 | 1 | 1 | 0 | | | $r_1$ | | $\neg w$ | $\neg x$ | $\neg y$ | $\neg z$ |
| x | x | x | 0 | 1 | 1 | 1 | 0 | | | | $r_0$ | $\neg w$ | $\neg x$ | $\neg y$ | $\neg z$ |

Table 3.2: Truth table and CNF clauses for a 3-input look-up table with SEUs

The truth table presented in Table 3.2 contains some rows that have *don't care* values (represented by x's) assigned to the input variables. In these rows, the output of the truth table ($z$) is assigned that value regardless of the value assigned to the variables that are assigned x's. When creating CNF clauses that represent these rows of the truth table, variables that are assigned x's are omitted from the clause.

For each $N$ input LUT with an SEU, $2N+2$ variables are used, although some of these are shared with other LUTs. Each LUT with an SEU also introduces $(N+2)2^N$ clauses. Although this is increases exponentially with the number of LUT inputs, this is not a concern since the number of inputs per LUT is bounded by the FPGA platform (usually $N \leq 6$). The total number of clauses in a circuit with $M$ LUTs would be $M(N+2)2^N$. This scales linearly with the number of LUTs in the design if $N$ is bounded.

Once these clauses have been produced for all LUTs in the function logic, constraints must be added such that only one LUT has a fault in it. Referring again to Table 3.2, variables $r_2 - r_0$ are common to all LUTs in the design. Variable $t$ is different for each LUT, and we will refer to the $t$ variable for LUT $i$ as $t_i$ where $0 \leq i < M$ and there are $M$ LUTs in the design. Since $t_i$ indicates whether LUT $i$ has an SEU, it is necessary that one of the $t_i$ is a one and the rest are zero. To that end, we introduce variables $u_0$ to $u_j$, where $j = log_2(M)$. The binary value held by variables $u_0$ to $u_j$ indicates which LUT in the design has an SEU in it, and clauses are added such that $t_i$ is equal to one when the binary value of $u_2$ to $u_0$ is $i$. An example is given in Table 3.3 for $t_3$, which indicates if LUT three has an SEU in a system that is composed of $5 \leq M \leq 8$ LUTs. If the binary value of $u_2$ to $u_0$ is three, then $t_3$ is one. Otherwise, $t_3$ is zero.

For a system with $M$ LUTs, the number of $t_i$ variables introduced here is $M$ and the number of $u_i$ variables is $log_2(M)$. The number of clauses per LUT is $log_2(M)+1$, and the total number for the system is $M(log_2(M)+1)$, so this scales well with the number of LUTs in the design.

| $u_2$ | $u_1$ | $u_2$ | $t_3$ |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
| 1 | x | x | 0 |
| x | 0 | x | 0 |
| x | x | 0 | 0 |

| | | | |
|---|---|---|---|
| $u_2$ | $\neg u_1$ | $\neg u_0$ | $t_3$ |
| $\neg u_2$ | | | $\neg t_3$ |
| | $u_1$ | | $\neg t_3$ |
| | | $u_0$ | $\neg t_3$ |

Table 3.3: Truth table and CNF clauses for $t_3$, which indicates if LUT three has an SEU

### 3.2.2 Flip-Flops

Realistic applications of FPGA-based systems require designs that use flip-flops. When flip-flops are included in the design, errors can cause the design to enter an incorrect state, even though an incorrect output has not yet occurred. For designs with flip-flops, the flip-flops are separated from the combinational logic of the design, and only the combinational logic of the design is analyzed. The flip-flop inputs are considered to be primary outputs of the combinational logic of the design, while the flip-flop outputs are considered to be primary inputs.

The flip-flops in FPGAs are generally designed with a set of inputs including enable, synchronous clear, synchronous load, and synchronous data in addition to the regular data input, as shown in Fig. 2.1. The part of the flip-flop hardware that computes the value actually stored in the flip-flop on each clock cycle can be considered to be part of the combinational logic of the design. Consider the circuit diagram of a flip-flop presented in Fig. 3.1. To avoid adding primary outputs to the

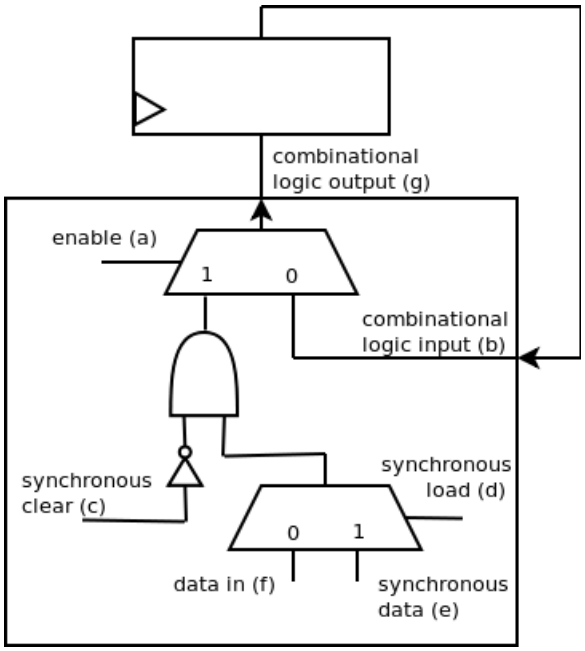


Figure 3.1: Circuit diagram of a flip-flop.

circuit for all of these flip-flop inputs, the flip-flop can be separated into a part that contains combinational logic, and a part that is state. The combinational part would then be represented in our CNF formula along with the LUTs that make up the design.

| a | b | c | d | e | f | g |
|---|---|---|---|---|---|---|
| 0 | 1 | x | x | x | x | 1 |
| 0 | 0 | x | x | x | x | 0 |
| 1 | x | 1 | x | x | x | 0 |
| 1 | x | 0 | 1 | 1 | x | 1 |
| 1 | x | 0 | 1 | 0 | x | 0 |
| 1 | x | 0 | 0 | x | 1 | 1 |
| 1 | x | 0 | 0 | x | 0 | 0 |

| | | | | | | |
|---|---|---|---|---|---|---|
| $a$ | $\neg b$ | | | | | $g$ |
| $a$ | $b$ | | | | | $\neg g$ |
| $\neg a$ | | $\neg c$ | | | | $\neg g$ |
| $\neg a$ | | $c$ | $\neg d$ | $\neg e$ | | $g$ |
| $\neg a$ | | $c$ | $\neg d$ | $e$ | | $\neg g$ |
| $\neg a$ | | $c$ | $d$ | | $\neg f$ | $g$ |
| $\neg a$ | | $c$ | $d$ | | $f$ | $\neg g$ |

Table 3.4: CNF for a flip-flop with synchronous clear, synchronous set, synchronous data, enable, and data input signals.

The truth table and CNF clauses for a flip-flop with combinational logic as presented in Fig. 3.1 are presented in Table 3.4. The output of the state part of the flip-flop (labeled with variable 'b' in Fig. 3.1 and Table 3.4) becomes a primary input to the combinational logic of the design. The input to the state part of the flip-flop (labeled with variable 'g' in Fig. 3.1 and Table 3.4) becomes an output of the combinational logic of the design. Variables a, c, d, e, and f are driven by other combinational logic.

### 3.2.3 Other FPGA Resources

In addition to flip-flops and LUTs, FPGAs are designed with a variety of other specialized hardware resources including embedded RAMs and multiplier blocks. These elements are not represented in the CNF formula and can be removed in a manner similar to how we deal with flip-flops. The output of these resources can become primary inputs to the combinational logic of the design, while their outputs can become primary inputs to the combinational logic of the design. In this way, we examine only the combinational logic and ignore faults that occur in these other hardware blocks. Many FPGA-based designs can be implemented with only LUTs and flip-flops, so we can disregard these other blocks for now significantly limiting our work, and fault detection in them could be a direction for future work.

## 3.3 Designing a Parity Check Matrix using QBF

Now that we have established methods for generating CNF clauses which represent the design, we will discuss some QBF-based methods for finding a parity check matrix

that is sufficient to detect all faults in the design. The idea is to represent the design with concurrent error detection logic using CNF clauses, and then add some additional clauses that represent properties that must be satisfied for the circuit to be fault secure. A method for representing the function logic with SEUs has been presented in Section 3.2, so we now move on to representing the redundant (parity predict) logic.

### 3.3.1 Representing the Parity Predict Logic and the Checker

The parity predict logic computes the parity bits of selected outputs as specified by the parity check matrix, $H$. The function logic is represented in a CNF formula with a set of clauses that constrain variables to follow the behaviour of signals in the design. When representing LUTs in the function logic, constraints are added such that there is a fault in only one of the LUTs (see Section 3.2.1). For the parity predict logic, LUT faults are also possible and can cause parity check bits to be incorrect. Since each parity group is checked by just one parity check bit, it is guaranteed that if any number of parity check bits are incorrect due to a fault in the parity predict logic, the incorrect behaviour will be detected by the checker. Although a fault could affect any number of parity check bits (since they are allowed to share logic), the fault could not simultaneously affect the function logic (since it is already optimized and cannot share logic with the parity check bits). For this reason, there is no need to be concerned with faults in the parity predict logic; the structure of our system guarantees that they will be detected.

When designing the full CNF formula including parity predict logic, we add a second set of clauses that represent the function logic but without faults. The input

variables are the same set for both the function logic with SEUs and this second set of clauses that does not contain SEUs, but all intermediate and output signals are represented by a second set of variables. The parity check bits will be computed (based on a parity check matrix) by taking the parity of selected outputs of this fault-free function logic. This does not represent a design that uses duplication; the parity check logic will be optimized as a whole and we expect it to be smaller than the function logic. We use duplication of the function logic in the CNF formula for the purposes of constraining the logical behaviour of the parity check bits.

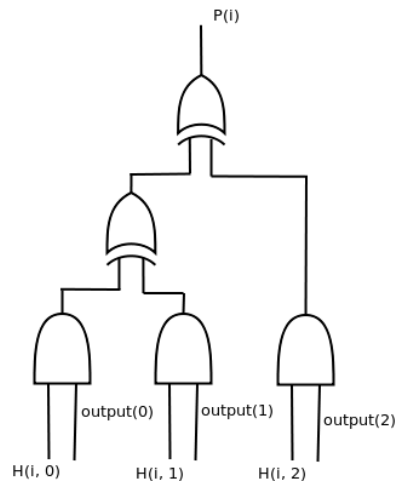


Figure 3.2: Diagram of the logic required to compute parity check bit $i$

A diagram of the logic required to compute the parity check bit $i$ of a circuit with three outputs is included in Fig. 3.2. A logical AND is computed of each output and the corresponding entry in the parity check matrix $(i, output)$, and the XOR of these results gives parity check bit $i$. Additional variables are added to represent the intermediate stages of the logic. The logical AND gives the value of the output if the the corresponding entry in the $H$ matrix is one, and a zero if the entry from the $H$ matrix is zero. In this way, the output has no impact on the computed parity bit if

the $H$ matrix is zero at that entry, and the output is included in the parity check if the $H$ matrix is one. The CNF clauses for the logical AND and XOR operations are generated using the normal procedure for converting truth tables to CNF clauses as previously described. A block diagram illustrating the structure of the CNF formula (including the clauses which represent the parity predict logic) is presented in Fig. 3.3

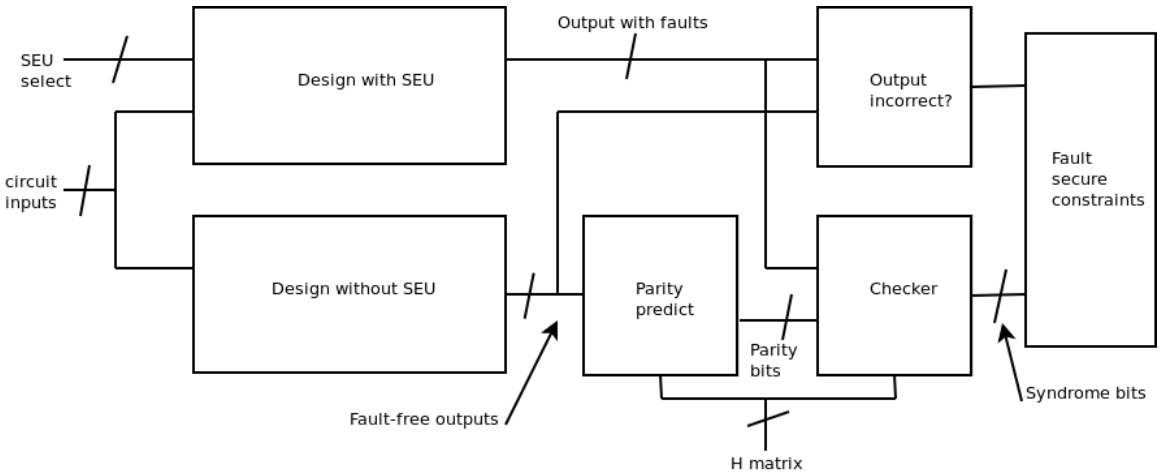


Figure 3.3: A block diagram illustrating the structure of the CNF formula

The checker is represented similarly to the parity predict logic. In the parity predict logic, we are taking the parity of a fault-free representation of the function logic to get the correct logical behaviour of the parity check bits. The checker computes the actual parity of the function logic outputs, but in this case, the outputs used are the outputs from the function logic that contains SEUs, so these parity bits could be incorrect if a fault is present in the design. The logical XOR is computed of each of the parity predict bits (from the redundant logic) and the corresponding parity bit from the design with SEUs, and the resulting bits are referred to as the *syndrome bits*. If a syndrome bit is one, that indicates that a fault is detected in the design, and if all syndrome bits are zero, no fault is detected. The clauses which represent

the checker are also shown in Fig. 3.3.

### 3.3.2 Representing Fault Secure Constraints

The circuit is *fault secure* if every incorrect output due to a fault is also a noncode word (Smith and Metze, 1978). The design of our redundant logic and checker guarantees that the syndrome bits will all be zero if there is no fault in the design. If there is a fault, one of the syndrome bits must be one if the fault causes an output to be incorrect. Since we would like to represent this requirement with CNF clauses, the first step is to generate clauses that give an indicator if an output is incorrect. The idea is to compute the logical XOR of each output from the function logic with SEUs with the corresponding output of the fault-free function logic, and the result will indicate if that output is incorrect due to a fault. The logical OR of all these results will indicate if there is any output that is incorrect as a result of a fault, and we will assume that this is represented by variable $e$.

Consider a design that uses three parity check bits which are computed by the redundant logic and three syndrome bits $s_2$-$s_0$. The syndrome bits can be constrained with the single clause $(s_2 \vee s_1 \vee s_0 \vee \neg e)$. This clause means that either one of the syndrome bits must be one, or the output error indicator ($e$) must be zero. Although it is also desirable to prevent the situation where there is no error indicated and yet a syndrome bit is high (an error is indicated even though no output is wrong), it is not necessary to include clauses that prevent this from happening since the way the design is constructed prevents this from happening. The fault secure constraints are shown within the structure of the CNF formula in Fig. 3.3.

### 3.3.3 Finding a Parity Check Matrix

Now that we have fully specified the CNF clauses, we examine the prefix (quantification) of the QBF instance. In general, we are trying to find a parity check matrix such that for all circuit inputs and fault locations the CNF clauses are satisfied. The prefix for the QBF instance should read: $\exists$ (variables representing the values in the $H$ matrix) $\forall$(variables representing all circuit inputs and fault locations) (all CNF clauses). If the instance is satisfiable, the QBF solver should return an $H$ matrix. If the solver returns 'UNSAT', then it is impossible to find a parity check matrix with the given number of parity check bits that can be used to create a fault secure circuit. In this case, the number of parity check bits could be increased, and the solver run again.

### 3.3.4 Feasibility of Using QBF

| Operand bits | 3-inputs LUTs | Parity bits | Solver result | Solver run time (hh:mm:ss) |
|---|---|---|---|---|
| 4 | 9 | 2 | UNSAT | 00:00:14 |
| 4 | 9 | 3 | SAT | 00:00:13 |
| 5 | 11 | 2 | UNSAT | 00:02:06 |
| 5 | 11 | 3 | SAT | 00:10:38 |
| 6 | 13 | 2 | UNSAT | 00:22:50 |
| 6 | 13 | 3 | SAT | 04:19:22 |
| 7 | 15 | 2 | UNSAT | 07:58:12 |
| 7 | 15 | 3 | SAT | 17:03:40 |

Table 3.5: QBF run times for a simple adder of varying size

Although a QBF solver can be used to find a parity check matrix that uses a specified number of parity check bits and results in a fault secure circuit, the approach has some significant limitations. When designing a parity check matrix, the goal is to

minimize the logic resources required to implement the redundant logic. The QBF-based approach does nothing to take into account the logic resources required by the parity check matrix that is selected. The most significant limitation, however, is the complexity of QBF solvers. Experiments were conducted using very small adder circuits, and the results are presented in Table 3.5. Solver run times are found using the QBF solver DepQBF (Lonsing and Biere, 2010). It is clear from these results that there would be significant run time concerns for circuits with use thousands of LUTs, so a QBF-based approach is impractical for realistic instances.

## 3.4 SAT-based Approaches

Using a QBF-based approach to designing a parity check matrix has significant run time and memory concerns, so it is not feasible for circuits that use thousands of LUTs. Instead, we look at SAT solvers to see how they can be used in an approach to this problem. When using a SAT solver, one cannot have existentially and universally quantified variables, as can be done with the prefix for QBF instances. Instead, one can consider a SAT instance to be a QBF instance with all variables existentially quantified. By changing a few CNF clauses, a SAT solver can still be of use.

### 3.4.1 Fault Secure Constraints for SAT

Using a SAT solver with the CNF clauses we have developed for use with QBF would result in a variable assignment which satisfies all clauses. Looking specifically at the constraint related to output errors and syndrome bits, a satisfying variable assignment would be one that causes either a syndrome bit to be high, or no output error to

occur. A SAT solver would find a single input vector and fault location that causes this to happen since we cannot specify that we want this condition to be satisfied for all inputs and all faults. In designing a CNF formula for use with a SAT solver, we change the constraints such that the solver looks for a variable assignment that indicates that the circuit is not fault secure. For a system with syndrome variables $s_2$-$s_0$ and variable $e$ indicating whether there is an incorrect output, unit clauses are added that constrain $s_2$-$s_0$ to be zero and $e$ to be one. In this way, we are looking for a variable assignment that causes an output to be wrong but is undetected by any syndrome bit.

### 3.4.2 Uses of SAT

This CNF formula is useful in a couple ways. If the $H$ matrix is constrained to a particular value using unit clauses, the SAT solver can be used to verify the fault secure property of the circuit. If the SAT solver returns 'UNSAT', then there is no variable assignment which causes an output error but is not detected by the syndrome bits. This means that the circuit is proven to be fault secure. If the SAT solver returns 'SAT', then there is a variable assignment which causes an error in a circuit output but is not detected by the syndrome bits. This means that the circuit is not fault secure. The satisfying variable assignment that is returned by the solver gives the input vector and fault location which was undetected. It also gives a list of outputs which were incorrect as a result of the fault, and this information can be useful for designing the parity check matrix.

The second way the SAT solver can be useful is in the analysis of which outputs can be affected by faults. If two outputs are checked by the same parity bit and they

are both incorrect, the fault will not be detected. When we are designing a parity check matrix, it is necessary to know which outputs can be incorrect at the same time. We define a *common fault vector* to be a subset of the circuit outputs which can be incorrect at the same time due to a LUT fault. The *common fault set* is a list of all common fault vectors.

The SAT solver can be used to extract the common fault set for a design. Recall that an all-solutions SAT solver is one which finds all satisfying assignments to a subset of variables in the CNF formula, and consider the set of variables in the CNF which each represent whether the corresponding circuit output is different from the correct value. If this set of variables is considered the subset of interest for an all-solutions SAT solver with an all zero $H$ matrix, the solver will return a list of all subsets of the circuit outputs which can be incorrect due to a single fault in the design (the common fault set). If this is done for a non-zero $H$ matrix, then a result of 'SAT' will return a list of all common fault vectors that are not caught by the current $H$ matrix.

## 3.5 Proposed Procedure

The proposed procedure involves using an all-solutions SAT solver to extract the common fault set for the design. The all-solutions SAT procedure is most similar to the one used in (Kang and Park, 2003). The solver is run iteratively, and every time an undetected fault is discovered, the common fault vector is added as a blocking clause so that we do not find the same one again.

### 3.5.1 Run Time Improvements

The speed with which SAT solvers determine the satisfiability of the CNF formula is greatly influenced by the number of clauses and variables in it. Instances of smaller size tend to execute more quickly. For this reason, we take a look at ways to reduce the size of SAT instances.

If we examine the LUT network which represents the combinational logic of the design, we can make some changes to the CNF formula which minimize its size. The goal at this stage is to identify groups of circuit outputs which can be incorrect at the same time as a result of a single LUT fault. It is only possible for a LUT fault to affect a circuit output if the LUT is included in the input cone of that output. If we examine faults in a particular LUT, we can ignore any output that does not have this LUT in its input cone. Only the set of outputs that contain this LUT are considered, and any LUT that is not part of the input cone for one of these outputs is ignored. We iterate through all the LUTs in the design and for each one we add constraints such that the fault occurs in that particular LUT. Once the solver returns 'UNSAT', that means there are no more undetected faults for this LUT, and we move on to the next one.

### 3.5.2 Extra Circuit Outputs

A problem that can occur during this stage is that a certain LUT is encountered which generates a very large number of common fault vectors when it contains an SEU. Consider an N-bit general purpose register in an ALU that has some logic connected to its synchronous clear input. The output to the combinational part of each of the flip-flops would become a primary output to the combinational logic of the

design. If a fault causes the synchronous clear port to be high when it is supposed to be low, the entire register will output zeros for each bit. Any bit of the register which would have been high in a fault free setting will now be incorrect. Since a general purpose register could hold any value on $(0, 2^N - 1)$, the total number of common fault vectors generated by this fault would be $2^N$. In this case, it is better to just duplicate the logic for the synchronous clear port, rather than try to store all these vectors and check them with a parity check code.

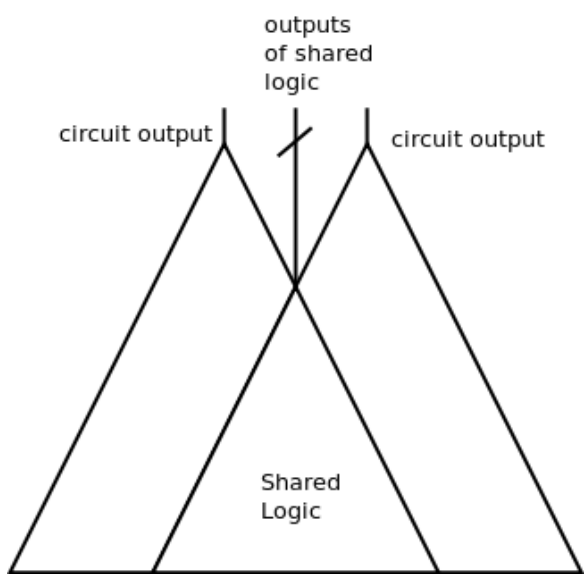


Figure 3.4: Diagram showing logic shared between two outputs.

Although there is a benefit to checking internal signals of the function logic, a mechanism for doing that has not yet been provided since the parity check matrix only includes outputs from the design. The solution provided is to select certain internal signals from the design and add them as outputs to the combinational logic so they can be checked using parity check bits. Consider the diagram in Fig. 3.4 that shows a circuit with two outputs and some logic shared between them. We would like to add the outputs of this shared combinational logic as primary outputs to the design

so they can be checked by the parity check bits. In general, every LUT in the design is included in the input cone of a subset of the circuit outputs. Each LUT output also has a fanout that may include other LUTs or flip-flops. Every LUT output, $A$, is added as a primary output to the circuit if there exists some LUT or flip-flop, $B$, in the fanout of $A$ such that the number of circuit outputs that have $B$ in their input cone is less than the number of outputs that have $A$ in their input cone.

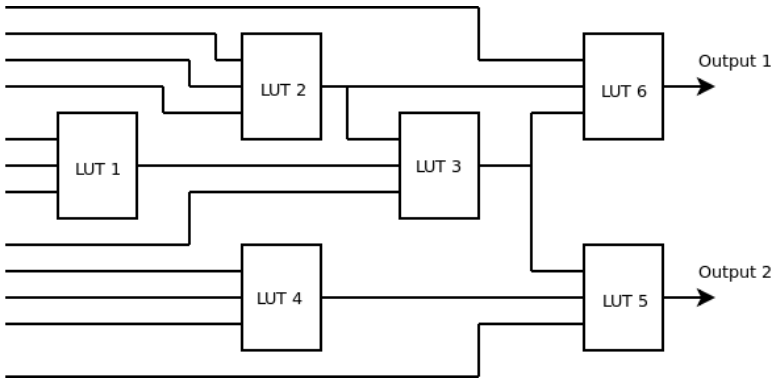


Figure 3.5: An example LUT network.

Consider the example LUT network presented in Fig. 3.5. LUTs 1, 2, and 3 are in the input cone of both outputs. LUT 6 is in the input cone of only output 1, while LUTs 4 and 5 are in the input cone of only output 2. LUTs 1, 2, and 3 make up the shared logic between the two circuit outputs. Consider LUT 1, which is in this shared logic. Its fanout is only to LUT 3, and LUT 3 is also in the input cone of both outputs, so the output of LUT 1 is not added as an additional circuit output. Consider now LUT 2 which is also in the input cone of both circuit outputs. The fanout of LUT 2 includes both LUT 3 and LUT 6. LUT 6 is in the input cone of only output 1, so the output of LUT 2 is added as an additional circuit output. The output of LUT 3 is also added as an additional output since it fans out to LUTs 4 and 5, and each of these are in the input cone of only one circuit output. In this LUT

network, the outputs of LUTs 2 and 3 are considered to be the outputs of the logic that is shared between outputs 1 and 2. Any fault occurring in LUTs 1, 2, or 3 must propagate through the output of either LUT 2 or 3 if it is going to affect either of the circuit outputs.

### 3.5.3 Duplicating Internal Signals

Now that we have established a group of internal signals that will become primary outputs to the design, we can move on to the original problem of duplicating internal signals which cause a large number of common fault vectors. A simple threshold is used to identify these problematic LUTs. Whichever circuit output appears most in these common fault vectors is added for duplication in the $H$ matrix. This means that a row is added in the $H$ matrix which includes only this output. Going forward, the common fault vectors that are extracted will be ones that are not caught by the current parity check matrix, so they will not contain this output in them.

## 3.6 Conclusion

This chapter has described an approach for creating a CNF formula that can be used to represent a circuit with SEUs in LUT configuration memory. A SAT solver can then be used to extract the common fault set, the set of all output groups that can be wrong at the same time due to a fault in the design. This set is proven to be complete due to the formal approach that has been used. The common fault set represents sufficient information about the design to generate a parity check matrix that can detect all faults and be used to form a fault secure circuit. Generating such

a parity check matrix is discussed in detail in Chapter 4.

# Chapter 4

# Selecting a Parity Check Matrix

In the previous chapter, we defined a *common fault vector* to be a group of all circuit outputs that can be incorrect at the same time due to a LUT fault in the design and an input vector. The *common fault set* was defined to be the set of all possible common fault vectors. When using a CED scheme that uses a systematic code to check circuit outputs, the design of the code must be such that it can detect all possible common fault vectors to guarantee a fault secure circuit. Recall that we have defined a *parity group* to be the group of outputs that are checked by some parity bit. For schemes that use a parity check code, there must be a parity group that contains an odd number of the outputs in each common fault vector in the common fault set.

This chapter will present an algorithm for generating a parity check matrix that can be used to create a fault secure circuit. Section 4.1 will describe some key ideas and insights that shape the approach taken in solving this problem. Section 4.2 will give a detailed description of the final algorithm.

## 4.1 General Approach and Rationale

The prior work on the subject of concurrent error detection based on parity check codes involves placing structural constraints on the function logic that guarantee that the selected code will detect all possible faults. The approach taken by (De *et al.*, 1994) involves partitioning the outputs into groups called *output groups*, where each output appears in exactly one output group. The logic that computes the outputs in an output group is optimized separately from other output groups, so that there is unconstrained logic sharing within an output group, but no logic sharing between output groups. A parity group can have at most one output from an output group. Since logic sharing is allowed within output groups but not between output groups, a single fault can affect only the outputs in one output group. In this way, each parity group can have only one incorrect output in it, and the parity check code is guaranteed to catch all faults that cause incorrect outputs. The results presented find that, in general, logic resource usage is minimized when there is a single output in each output group and a single parity bit is used.

The approach taken by (Touba and McCluskey, 1997) introduces a procedure called structure-constrained logic optimization (SCLO). Using SCLO, logic optimization can be done that honours structural constraints with regards to logic sharing between circuit outputs. A parity check code is selected using an algorithm that estimates the logic resources required by different codes. Each output appears in exactly one parity group. Within a parity group, there can be no logic sharing between any two outputs or any output and the parity bit itself, but logic sharing is allowed between outputs that are in different parity groups, between parity bits, and between parity bits and outputs not in the related parity group. In this way, a minimal set

of structural constraints are generated. These constraints are used during SCLO to optimize the entire circuit (function logic and parity predict logic) at once.

The results published in (Touba and McCluskey, 1997) indicate that this approach produces circuits that use fewer logic resources than those produced by the technique presented in (De *et al.*, 1994). This result is intuitive; a technique that places fewer constraints on logic optimization should allow the multi-level synthesis tool to produce better results. Although our approach is fundamentally different than the one proposed in (Touba and McCluskey, 1997), we identify it as the most successful and draw inspiration from it in this section.

### 4.1.1 Design Flow Benefits and Limitations

The structural constraints used in (Touba and McCluskey, 1997) do not allow a parity bit to share logic with outputs that are in the associated parity group, but logic sharing is allowed with the other outputs in the function logic. Our approach involves unconstrained logic optimization and LUT mapping of the function logic, designing a parity check code, and optimizing the parity predict logic separately from the function logic. In this way, the design flow that is used eliminates logic sharing between the parity predict logic and the function logic, thus creating a structural constraint between function logic outputs and parity check bits that may not be necessary to satisfy the fault secure property. The disadvantage of this approach is that there may be logic that could be shared between some parity bits and function logic outputs, resulting in a design that uses fewer logic resources, and this benefit cannot be realized.

The advantage in using an approach that optimizes the function logic separately

from the parity predict logic is that our approach can use available LUT mapping procedures that do not necessarily honour the more complicated structural constraints present in (Touba and McCluskey, 1997). Another benefit is that it enables the proposed formal approach, which allows us to exploit the particular realities of how faults affect the optimized function logic, rather than making potentially unnecessary assumptions based on its structure. Instead of assuming that faults can act a certain way based on the structure of the circuit, we look how faults can actually behave using a formal approach. This has the potential to enable logic sharing that would not have been possible if using the technique presented in (Touba and McCluskey, 1997).

### 4.1.2 Resource Estimation

Part of the reason for the success of the approach presented in (Touba and McCluskey, 1997) is that it presents a method for estimating the resource usage of different parity check codes. The approach identifies ways that changing the parity check code affects the size of the optimized logic. The resources required to implement the function logic is affected by the parity check matrix because different parity check codes result in different logic sharing constraints. The parity check code also affects the size of the checker which computes the actual parity of the function logic outputs. Last, the parity check code affects the size of the parity predict logic.

Although we expect that, in general, parity check codes with fewer parity bit will result in fewer logic resources to implement the parity predict logic, this is not necessarily the case. Different codes with the same number of parity bits may also lead

to different resource counts once fully optimized. The approach in (Touba and McCluskey, 1997) uses a logic optimization tool to perform two-level logic optimization of individual parity bits. The logic optimization tool returns the literal count of the logic formula in two-level (sum-of-products) form. Although this does not take into account the resource sharing benefits that may be realized by doing a full multi-level synthesis, it provides a quicker way to estimate the resource usage of different parity check codes.

The approach used in this thesis borrows from the techniques presented in (Touba and McCluskey, 1997) by performing a logic optimization of parity check bits as a method of estimating the area of the parity predict logic. Instead of performing a two-level optimization (which would require collapsing of the Boolean network formed by the optimized and LUT-mapped function logic), we instead perform a multi-level optimization of just the parity bit for which we intend to estimate the area. For a multi-level synthesis, the Boolean logic formula is put in a factored-form that gives a multi-level representation (Brayton *et al.*, 1990). In this case, the tool returns the *factored-form literal count*, which is the number of literals in the Boolean formula in the factored-form after multi-level optimization, and this is taken as an estimate of the resources required by the logic. A full description of multi-level synthesis is beyond the scope of this thesis.

The logic required for the XOR network used to implement the checker can be calculated, so the use of a logic optimization tool is not required. Although no LUT mapping is performed as part of the resource estimation, we assume that the factored form literal count is an accurate method for comparing different parity check codes, even though it does not tell us what the final LUT usage will be. For the function

logic, logic optimization (and LUT mapping) is performed prior to designing the parity predict logic, so the size of the function logic does not change depending on which parity check code is used.

### 4.1.3 Logic Sharing Constraints

Part of the the resource estimation approach of (Touba and McCluskey, 1997) involves examination of the trade off between parity predict logic and function logic resources based on the number of parity bits that are chosen. The algorithm used in (Touba and McCluskey, 1997) starts with a parity check code that is equivalent to duplication and improves it by examining which parity groups can be merged to create a parity check code that uses fewer logic resources. When examining the possibility of merging two parity groups, part of this is weighing the potential improvement in logic resources required for the parity predict logic against the cost incurred by the added structural constraints that are needed since there are more outputs that cannot share logic. If the improvement to the parity predict logic is greater than the cost incurred due to additional structural constraints, then the parity groups are merged and one fewer parity bits is used.

The affect of the additional logic sharing constraint between two outputs is estimated by assuming that any logic shared by these outputs must be duplicated (Touba and McCluskey, 1997). This is a worst case scenario; there maybe other logic sharing opportunities that can be exploited as a result of the restructuring of the circuit. In the approach presented in this thesis, the function logic is optimized prior to selecting a parity check code, so structural constraints cannot be added to enable putting outputs that share logic in the same parity group.

Section 3.5.2 identifies a technique for taking the signals which represent the output of shared logic cones, and adding them as outputs to the design. Consider Fig. 3.5 from section 3.5.2. LUTs 1, 2, and 3 make up the logic resources that are shared between both outputs of the design. If these outputs are to be checked by the same parity bit, the approach presented in (Touba and McCluskey, 1997) would add a constraint to prevent logic sharing between these outputs, and this would result in duplication of the entire logic cone (LUTs 1, 2, and 3) in the worst case. In our approach, the output of LUTs 2 and 3 are added as outputs that can be checked by the parity bits. Any fault in LUT 1, 2, or 3 that causes both outputs 1 and 2 to be incorrect will propagate through the output of either LUT 2 or 3 (or both). If a common fault vector exists which indicates that two outputs can be wrong at the same time, the vector will also contain some of these internal signals which affect both outputs and through which the error has propagated. By duplicating these internal signals in the parity check matrix, the outputs can now be checked in the same parity group, since these internal signals are used to detect the vector which contains both outputs. The affect of this is similar to introducing logic sharing constraints as in (Touba and McCluskey, 1997).

Duplication is a worst case result of checking internal signals in the parity check matrix. These signals are synthesized as outputs of the parity predict logic, so there are logic sharing opportunities with the other parity bits in the design. In addition, duplication is not the only way of checking these signals; they can also be checked as part of a larger parity group, so that logic can prove to be smaller than duplication.

### 4.1.4 Essential Outputs

As discussed in section 3.5.2, some signals from inside the design are treated as outputs so they can be checked in the parity check matrix. The result is that some of the outputs in each common fault vector do not represent original outputs from the circuit but represent internal signals in the design. Since these signals do not represent actual outputs, they do not need to be checked to guarantee a fault secure circuit. We design the CNF formula to only look for common fault vectors that contain at least one output of the original design.

Any output that appears as the only output in a common fault vector must be checked somewhere in the parity check matrix to make sure that this common fault vector is detected. We refer to these as *essential outputs* since they must exist in the parity check matrix somewhere. Any essential output is also an output from the original design, rather than an internal signal, because the CNF formula is designed such that each common fault vector contains at least one output of the original design. In general, most original outputs are essential outputs since there usually exists some fault that can cause that output to be the only incorrect one.

### 4.1.5 Algorithm Overview

The algorithm used for selecting an H matrix involves starting with an empty matrix and applying permutations to it until the matrix is able to detect all common fault vectors. Each permutation to the H matrix involves selecting an output (column in the H matrix) and parity group (row in the H matrix), and replacing the entry at that position with its complement. The goal in selecting outputs is to choose ones that will result in permutations that catch common fault vectors that contain that

output. A common fault vector is detected if there is some row of the parity check matrix that contains an odd number of the outputs in that vector.

Consider the matrix presented in Fig. 2.1. The common fault vector $(1, 2, 3)$ would be caught by parity group two, but not by one or three. If entry $i, j$ is permuted (where $i$ is the row and $j$ is the column), any common fault vector that contains output $j$ and is not being detected by parity group $i$ will now be detected, while any common fault vector that contains $j$ and is being detected by $i$ would no longer be detected. If output one in parity group one is replaced by its complement, $(1, 2, 3)$ would now be detected by parity group one, while $(1, 3)$ would no longer be detected by parity group one. Every time a parity group is permuted at $i, j$, the result is that any common fault vector containing $j$ that was undetected, is now detected. Any common fault vector containing $j$ that was detected by $i$ is no longer detected by $i$. Any common fault vector that was detected *only by parity group* $i$, is now undetected.

Since essential outputs must be checked in the parity check matrix somewhere, we design our algorithm to place them into parity groups before moving on to the remainder of the outputs. Essential outputs are checked in the parity check matrix to detect the single output common fault vector in which they occur, however they are useful for detecting all common fault vectors they appear in. If an essential output is included in a parity group by itself (output duplication), then all common fault vectors which contain this output will be detected. Consider the effect of adding essential output $j$ into a parity group $i$. Any common fault vector that is detected by that parity group and contains $j$ is no longer detected, and this cost should be considered before making that decision.

This is analogous to the approach used in (Touba and McCluskey, 1997). When

two parity groups are merged to produce a single larger parity group, the effect of the logic sharing constraints is estimated by assuming duplication of any logic that is shared between two outputs. In our approach, this logic duplication occurs by checking internal signals of the design. When an essential output is added to a parity group, we examine the common fault vectors that are no longer detected as a result of this addition and estimate the cost of checking them with non-essential outputs. After selecting parity groups in which to include essential outputs, the algorithm uses a greedy approach to permute other locations in the parity check matrix until all common fault vectors are caught. The full algorithm is presented in the next section.

## 4.2 Parity Check Matrix Algorithm

In the previous section, we identified that the resource usage of a parity check matrix can be estimated by looking at the size of the resulting parity predict logic and checker. This section will introduce some functions that will be used to estimate logic resources before presenting the final parity check matrix algorithm.

### 4.2.1 `AREA(i,j)`

The previous section introduced a method for estimating logic resources using two-level logic optimization. This function estimates the change in logic resources of the parity predict logic and checker that occurs due to permuting entry $i, j$ of the parity check matrix, where $i$ is the parity group and $j$ is the circuit output. The logic resources required by a parity check bit is estimated by performing a two-level optimization of that bit using MVSIS v3.0. The checker consists of an XOR network

that computes each parity bit and an OR network that combines the syndrome bits into a single error indicator. If $N$ is the number of outputs in a parity group, the number of XORs required to compute the corresponding syndrome bit is $N$. For a design with $M$ parity groups, the number of ORs required to combine the syndrome bits to get the error indicator is $M-1$. The factored-form literal count of an XOR is four and for an OR it is two, so $4*N+2*(M-1)$ computes the factored-form literal count of the entire checker. The result returned from `AREA(i,j)` is an estimate of the increase in size of the parity predict logic and checker due to permuting the entry in the $H$ matrix at output $j$ and parity group $i$. The factored-form literal count for parity bit $i$ and the checker is found before and after permutation of entry $i,j$, and the difference between them is returned to give the change in literal count.

### 4.2.2 `SHARED(i,j)`

When an essential output is added into a non-empty parity group, we estimate the cost of detecting common fault vectors that become undetected as a result of this addition as described in Section 4.1.5. This is done by assuming that all non-essential outputs in these undetected common fault vectors may have to be duplicated (checked each in their own parity group) in the worst case to detect these vectors. If one of these common fault vectors contains some essential output that has not yet been added to the parity check matrix, we disregard it because we anticipate that it will be checked in the future via this essential output.

The function `SHARED(i,j)` generates a list of all common fault vectors that are undetected after adding output $j$ to parity group $i$. Any common fault vector that contains some essential output that has not yet been added to the parity check matrix

| | | Information Bit | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 |
| Parity Bit | 1 | 1 | 0 | 0 | 0 | 0 |
| | 2 | 0 | 0 | 0 | 0 | 0 |

| | | | |
|---|---|---|---|
| 1 | | | |
| 1 | 2 | 4 | |
| 1 | 4 | | |
| 1 | 2 | 3 | 5 |
| 2 | | | |
| 2 | 4 | | |
| 2 | 5 | | |
| 2 | 3 | 5 | |
| 3 | | | |
| 3 | 5 | | |

Table 4.1: An example $H$ matrix and common fault set

is discarded from this list. The function then looks at all non-essential outputs that appear in this list and performs a logic optimization of them using MVSIS v3.0. The function returns the factored-form literal count from the logic optimization.

Consider the example $H$ matrix and common fault set presented in Table 4.1. One is an essential output since it is the only output that appears in the first common fault vector, and 1 has already been added to the $H$ matrix in parity group 1. Output 2 is also an essential output since there is a common fault vector that contains only 2. Adding output 2 to parity group 1 would result in the detection of common fault vectors $(2)$, $(2,4)$, $(2,5)$, and $(2,3,5)$, but $(1,2,4)$ and $(1,2,3,5)$ would not be detected. We discard $(1,2,3,5)$ because it may be detected in the future due to output 3 also being an essential output. Examining the remainder, just vector $(1,2,4)$, the only non-essential output is 4, so the function call `SHARED(1,2)` would return the estimated factored-form literal count of duplicating output 4. If output 2 were added to parity group 2, the result is that all common fault vectors containing 2 would be detected. The function call `SHARED(2,2)` would return zero.

### 4.2.3 `CHANGE(i,j)`

After all the essential outputs have been added to the parity check matrix, a greedy heuristic is used that permutes entries of the matrix until all common fault vectors have been detected. Part of that involves calculating the change in undetected common fault vectors as a result of some permutation. The function `CHANGE(i,j)` returns the change in the number of detected common fault vectors as a result of complementing location $i, j$ of the parity check matrix.

Consider the example $H$ matrix and common fault set presented in Table 4.1. The $H$ matrix presented detects 4 common fault vectors: $(1)$, $(1, 2, 4)$, $(1, 4)$, and $(1, 2, 3, 5)$. If output 5 is added to parity group 1, $(1)$, $(1, 2, 4)$, $(1, 4)$, $(2, 5)$, $(2, 3, 5)$, and $(3, 5)$ would now be all the detected common fault vectors, while $(1, 2, 3, 5)$ would no longer be detected since it contains both 1 and 5. Since 4 common fault vectors were detected before the addition and 6 were detected after, `CHANGE(1,5)` would return $6 - 4 = 2$.

### 4.2.4 `PARITY_CHECK`

The full algorithm for generating a parity check matrix, $H$, is presented in Fig. 4.1. The variable $n$ represents the number of parity groups, $H$ represents the parity check matrix, and $M$ represents the set of common fault vectors that are not detected by $H$. If there is an essential output that does not appear in a parity group, then the unit vector it appears in will be undetected. Line five checks for such a vector, and if one exists, the output is selected for addition (line 6). The algorithm first estimates the change in logic area if the output is added to a new parity group (lines 7,8). In this case, a call to `SHARED(i,j)` is unnecessary since all common fault vectors containing

```
1: n ← 0
2: M ← undetected common fault set
3: H ← parity check matrix
4: while M ≠ ∅ do
5:     if ∃m ∈ M, |m| = 1 then
6:         j ← m
7:         i ← n + 1
8:         area ← AREA(i,j)
9:         for k = 1 → n do
10:            if (AREA(k,j) + SHARED(k,j)) < area then
11:                area ← AREA(k,j)+SHARED(k,j)
12:                i ← k
13:            end if
14:        end for
15:    else
16:        j ← most undetected output
17:        i ← n + 1
18:        area ← AREA(i,j)/CHANGE(i,j)
19:        for k = 1 → n do
20:            if (AREA(k,j)/CHANGE(k,j) < area) ∧ (CHANGE(k,j) > 0) then
21:                area ← AREA(k,j)/CHANGE(k,j)
22:                i ← k
23:            end if
24:        end for
25:    end if
26:    H(i, j) ← ¬H(i, j)
27:    M ← undetected common fault set
28:    if i = n + 1 then
29:        n ← n + 1
30:    end if
31: end while
32: return n, H
```

**Figure 4.1:** PARITY_CHECK: The algorithm used for selecting a parity check matrix.

that output will be detected. The algorithm then searches through all other parity groups and estimates their area in an attempt to find one that requires fewer logic resources (lines 9-14).

If there is no undetected unit common fault vector, then the algorithm selects the output that appears in the most undetected common fault vectors for permutation (line 16). Instead of just looking at the change in area caused by placing the output in a particular parity group, the area estimation is divided by the change in detected common fault vectors to get an estimation of factored-form literals per vector. As before, the algorithm first gets an estimate of this for placing the output in a new parity group (lines 17,18) and then examines the other parity groups to see if there is one that offers an improvement.

Once a parity group $i$ and output $j$ have been selected, the $H$ matrix is complemented at $i, j$, and the undetected common fault set is updated (lines 26,27). If the output was added to a new parity group, the number of parity groups $n$ is incremented (lines 28-30). The algorithm terminates when there are no undetected common fault vectors.

# Chapter 5

# Presentation of Results

This chapter will present and discuss the results of using the proposed procedure on a set of benchmarks circuits and two small microprocessors. Results for circuits from the MCNC benchmark suite will be given in Sections 5.1 and 5.2. In evaluating the effectiveness of the technique, we will examine two prior techniques for concurrent error detection. The first, circuit duplication, is an obvious point of comparison since it is an established technique that is easily implemented for FPGA-based designs. Second, we look at the results presented by (Touba and McCluskey, 1997), since they represent the prior work that is closest to the proposed technique. Results for a microprocessor will then be given in Section 5.3.

## 5.1 Benchmark Results

The proposed procedure was first tested on a group of circuits from the MCNC benchmark suite (Yang, 1991). The function logic underwent synthesis and technology mapping using the Quartus II tool flow. The design was targeted for an Altera Cyclone

II FPGA, which has 4-input LUTs for implementing combinational logic functions. The resulting parity predict logic was optimized and mapped to LUTs using the Quartus II tool flow. The results are included in Table 5.1. Note that the results for the proposed procedure and for duplication are for the parity predict logic plus the checker, so that is why the LUT count is higher than for the original circuit.

| circuit name | original circuit LUTs | duplication LUTs | procedure LUTs |
|---|---:|---:|---:|
| 5xp1 | 30 | 34 | 34 |
| alu4 | 727 | 810 | 476 |
| bw | 54 | 80 | 60 |
| b12 | 21 | 29 | 22 |
| f51m | 28 | 40 | 41 |
| misex2 | 38 | 46 | 41 |
| misex1 | 19 | 23 | 27 |
| pcle | 20 | 25 | 24 |
| term1 | 78 | 92 | 91 |
| ttt2 | 50 | 67 | 55 |
| x2 | 15 | 19 | 16 |
| cmb | 16 | 18 | 11 |
| cu | 16 | 22 | 16 |

Table 5.1: The area results for benchmark circuits after logic optimization and LUT mapping.

The results are are favourable in the sense that the parity predict logic for many circuits shows an improvement over using a concurrent error detection scheme based on duplication. The best result (alu4) shows logic resources that are less than 60% of what was required for a scheme based on duplication. In the worst case, the number of LUTs required for the parity predict logic is not much worse than using duplication. The worst result (misex1) shows logic resources that are approximately 117% of what was required for a scheme based on duplication. This result is intuitive. A worst case implementation of some parity predict logic is circuit duplication plus an XOR network to compute the parity of the duplicated outputs. This would certainly be

worse than duplication (because of the XOR network), but we expect the logic to compute the parity to be small relative to the size of the duplicated function logic.

We consider this comparison to be favourable since there are a significant number of circuits for which the proposed technique offers an improvement over duplication. Duplication can be considered to be a subset of parity check codes where there are $N$ parity bits used to check $N$ outputs, and output $i$ is checked only by parity bit $i$. We are not overly concerned with the circuits where the proposed technique was worse than duplication, even though this would represent a failure of the proposed algorithm to find the parity check code that requires the fewest resources. The most important requirement on an algorithm that selects a parity check code is that it is able to find parity check codes that require fewer logic resource than duplication when this is possible for the circuit being examined. It is not important that the algorithm identify that duplication is the best possible result for circuits where this is the case. If the result that is returned is worse than duplication, duplication can just be used anyway since it is easily implemented without any additional tool support.

## 5.2 Comparison with Prior Techniques Using Parity Check Codes

Recall that in Section 4.1 we identified the technique from (Touba and McCluskey, 1997) as the most successful prior work in the area of designing concurrent error detection logic based on parity check codes. It was further established in Section 4.1.1 that our technique has fundamental limitations that are not present in (Touba and McCluskey, 1997) since we restrict any logic sharing between the function logic

and the parity predict logic. If there is some potential for logic sharing between the function logic and the parity predict logic, our approach cannot reap the benefits. On the other hand, this enables our formal approach. The formal approach enables us to determine precisely which outputs can be incorrect at the same time due to faults instead of assuming this based on the structure of the circuit. We attempt to quantify these affects by generating results which can be compared to those published in (Touba and McCluskey, 1997).

| circuit name | redundant circuit Lits | function logic Lits | Total Lits | Percent |
|---|---|---|---|---|
| 5xp1 | 133 | 134 | 267 | 102 |
| bw | 345 | 178 | 523 | 119 |
| b12 | 118 | 87 | 205 | 106 |
| f51m | 198 | 130 | 328 | 131 |
| misex2 | 161 | 104 | 265 | 95 |
| misex1 | 94 | 54 | 148 | 116 |
| pcle | 110 | 69 | 179 | 91 |
| term1 | 234 | 179 | 413 | 107 |
| ttt2 | 281 | 191 | 472 | 97 |
| x2 | 74 | 51 | 125 | 117 |
| cmb | 36 | 52 | 88 | 110 |
| cu | 79 | 53 | 132 | 107 |

Table 5.2: Results in terms of factored-form literals, and expressed as a percentage of the results in (Touba and McCluskey, 1997)

It is not possible to make a direct comparison between our results and those presented in (Touba and McCluskey, 1997). For the proposed approach, the logic resources are expressed in terms of the number of LUTs, since the design is to be implemented on an FPGA. The number of logic resources in (Touba and McCluskey, 1997) are expressed in terms of the number of literals (lits) that make up the factored form Boolean equations in the logic network. Performing a LUT mapping of this Boolean logic network would appear to be the ideal solution, but commercial LUT

mapping tools are not designed to honour the structural constraints that are required to guarantee the fault secure property (as discussed in Section 2.6).

Rather than attempting to design such a tool (which would be a considerable undertaking), we perform a multi-level optimization of the logic equations that represent the parity predict bits produced by our technique and report the logic resource usage in terms of literals. Table 5.2 gives the literal count for the parity predict logic that was generated for some circuits in the MCNC benchmark suite. Results were generated for the same circuits in (Touba and McCluskey, 1997). Recall that the technique in (Touba and McCluskey, 1997) involves simultaneous optimization of the parity predict logic and the function logic, so the generated results are for these combined. In Table 5.2, we give the literals required to implement the redundant logic and the function logic. The sum of these is presented in the fourth column for comparison to (Touba and McCluskey, 1997). In the fifth column, the total literals required for our procedure is presented as a percentage of the number of literals required for the procedure in (Touba and McCluskey, 1997).

The results presented in this thesis are similar to those presented in Touba and McCluskey (1997) in terms of literal count. On average, circuits with concurrent error detection that are produced using the techniques in this thesis use 108% of the literal count of those presented in (Touba and McCluskey, 1997). This indicates that the limitation imposed by restricting logic sharing between the parity predict logic and the function logic slightly outweighs the benefits of the the formal approach in terms of resource usage.

## 5.3 Microprocessor Circuit Results

In Section 5.1, we demonstrated that the proposed technique compares favourably in terms of logic resources to techniques that use circuit duplication when applied to some combinational benchmark circuits. In Section 5.2, it was shown that the proposed technique comes close to producing the same resource usage results as were presented in (Touba and McCluskey, 1997), despite of some limitations imposed by the structure of the circuit. In this section we attempt to apply these techniques to a microprocessor circuit to see how it can be applied to a typical application.

| Circuit LUTs | Circuit Outputs | Parity Bits | Redundant Circuit LUTs | Run Time (hh:mm:ss) |
|---|---|---|---|---|
| 1362 | 479 | 271 | 1606 | 24:40:58 |

Table 5.3: Results of proposed procedure for an 8-bit AVR microcontroller.

The test circuit is an 8-bit CPU core from an AVR microcontroller, and results are presented in Table 5.3. The original design (function logic) requires 1362 LUTs to implement. The combinational logic of the design has 479 outputs, and the parity predict logic is found to use 271 parity bits. The parity predict logic is implemented with 1606 LUTs. The total run time for the algorithm was 24 hours, 40 minutes, and 58 seconds.

The results presented are not entirely positive, and some reasons for this will be suggested. Although the total run time is long (around 24 hours), we consider this to be acceptable. The tool flow proposed in this thesis needs to be run only once when the HDL code for the function logic has been implemented an verified. The redundant logic is generated only after the design process for the function logic is complete, so there is no need to apply the procedure iteratively as part of the design process. In comparison to the amount of time required to design and implement custom hardware

designs, 24 hours is insignificant.

The results generated are for the parity predict logic only (no checker). The number of LUTs required for the redundant circuit is greater than the original circuit, so this indicates that our approach finds a result that is worse than duplication. A few potential reasons for this will be presented. First, consider the combinational logic that is part of a flip-flop as presented in Section 3.2.2. This logic has been included with the combinational logic of the design. The hope is that when outputs are checked together in a parity bit, the resulting logic might be smaller than the sum of the outputs, and in this way, the combinational logic around a flip-flop might be reduced. This combinational logic is implemented as part of the flip-flop in the original design, so it does not get mapped to LUTs. If this combinational logic is not reduced when checked with a parity bit, it will be mapped to LUTs by the tool, causing an increase in the LUT count of the design.

Another reason that the proposed technique is using more LUT resources than duplication may be related to the way the FPGA is implementing some arithmetic functions. In Cyclone II FPGAs, binary adders are mapped to LUTs in a specific way. In arithmetic mode, 4-input LUTs can be divided into two 3-input LUTs. In that way, each 4-input LUT is used to implement a stage of a ripple-carry adder. One of the two 3-input LUTs is used to compute the sum bit, while the other is used to compute the carry. The Quartus II tool flow is able to identify adders in the HDL code and place them into LUTs in this way. If the combinational logic that represents an adder does not get reduced as part of a parity check bit, it may be mapped to LUTs inefficiently because the tool is not able to identify it as in adder in the HDL code.

# Chapter 6

# Conclusion

In this thesis, we have established an approach for designing concurrent error detection logic based on parity check codes for hardware designs that are implemented on FPGAs. A formal approach is used to guarantee that the resulting circuit is able to detect the presence of any SEU in a LUT configuration bit within one clock cycle of it producing an incorrect circuit output. In this way, the resulting circuit meets the robustness requirement set out in Chapter 1 of this thesis. Most circuits with concurrent error detection that are produced by the proposed technique require fewer LUT resources than circuits that use duplication. The technique has been shown to be scalable to designs that use over one thousand look-up tables.

For safety-critical systems, it is necessary for the designer to have a high degree of confidence in the tool flow that is used. Any new software tool that is added into the flow has the potential to introduce errors until if it is not properly verified. Many previous techniques for concurrent error detection rely on structural constraints during logic optimization, so a custom LUT mapping tool would have to be designed if these techniques are to be applied to FPGA-based designs. Our approach implements

concurrent error detection logic that is optimized separately from the original design logic. In this way, the original design can use an established, commercially available tool flow for logic optimization and LUT mapping. The concurrent error detection logic does not interfere with the function logic since it is optimized separately. In this way, errors in the tool that produces the redundant logic cannot cause the original design logic to function incorrectly.

## 6.1 Future Work

The techniques proposed in this thesis are useful for detection of faults in combinational FPGA-based designs that are mapped to look-up tables. Examining faults only in look-up tables just scratches the surface of designing FPGA-based systems with error detection capabilities. Some other possible directions for research are suggested in this section.

### 6.1.1 Integration of Arithmetic Techniques

In Section 5.3, we explored some reasons why the proposed approach might be unable to generate redundant logic for a microprocessor design that uses fewer resources than circuit duplication. One reason that was identified was that the LUT mapping tool has a specialized way of mapping of adders in the design to look-up tables. Prior work has identified multiple techniques for detecting errors in hardware that performs arithmetic operations (Gorshe and Bose, 1996; Lo *et al.*, 1989). An algorithm that integrates these approaches to efficiently check arithmetic hardware in a design could be used to improve the results.

### 6.1.2 Faults in other FPGA Components

Look-up table configuration bits form only a small fraction of the configuration bits in an FPGA that can be affected by SEUs. For designs with state machines, flip-flops can be directly affected by SEUs, corrupting the state of the circuit. Flip-flops and look-up tables are connected together by a programmable interconnect fabric that is configured through SRAM memory. If this configuration memory is corrupted by an SEU, the routing of the FPGA would be affected, thus changing the functional behaviour of the circuit. A technique for detecting interconnect faults is another possibility for future research.

## 6.2 Closing Remarks

The results presented in this thesis show that concurrent error detection logic can be added to many circuits without incurring the full cost of circuit duplication. Although some excellent results have been demonstrated, the procedure needs polishing to produce high quality results for all practical applications. Eventually, we would like to develop a fully automated tool flow that produces circuits that are proven to detect faults in all FPGA configuration bits while minimizing logic resources. This research provides an excellent starting point towards this goal.

# Bibliography

Altera Corporation (2007). Cyclone II device handbook. `http://www.altera.com/literature/hb/cyc2/cyc2_cii5v1.pdf`.

Altera Corporation (2008). Error detection and recovery using CRC in Altera FPGA devices. `http://www.altera.com/literature/an/an357.pdf`.

Avizienis, A. and Kelly, J. (1984). Fault tolerance by design diversity: Concepts and experiments. *Computer*, **17**(8), 67 –80.

Baleani, M., Ferrari, A., Mangeruca, L., Sangiovanni-Vincentelli, A., Peri, M., and Pezzini, S. (2003). Fault-tolerant platforms for automotive safety-critical applications. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03, pages 170–177, New York, NY, USA. ACM.

Balint, A., Belov, A., Diepold, D., Gerber, S., Järvisalo, M., and Sinz, C., editors (2012). *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of *Department of Computer Science Series of Publications B*. University of Helsinki. ISBN ISBN 978-952-10-8106-4.

Berger, J. (1961). A note on error detection codes for asymmetric channels. *Information and Control*, **4**(1), 68 – 73.

Bolchini, C., Salice, F., and Sciuto, D. (2002). Designing self-checking FPGAs through error detection codes. In *Defect and Fault Tolerance in VLSI Systems, 2002. DFT 2002. Proceedings. 17th IEEE International Symposium on*, pages 60 – 68.

Bose, B. and Lin, D. J. (1985). Systematic unidirectional error-detecting codes. *Computers, IEEE Transactions on*, **C-34**(11), 1026 –1032.

Brayton, R., Hachtel, G., and Sangiovanni-Vincentelli, A. (1990). Multilevel logic synthesis. *Proceedings of the IEEE*, **78**(2), 264 –300.

Das, D. and Touba, N. (1998). Synthesis of circuits with low-cost concurrent error detection based on bose-lin codes. In *VLSI Test Symposium, 1998. Proceedings. 16th IEEE*, pages 309 –315.

De, K., Natarajan, C., Nair, D., and Banerjee, P. (1994). RSYN: a system for automated synthesis of reliable multilevel circuits. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, **2**(2), 186 –195.

Feng, Z., Hu, Y., He, L., and Majumdar, R. (2009). IPR: In-place reconfiguration for FPGA fault tolerance. In *Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, pages 105 –108.

Feng, Z., Jing, N., Chen, G., Hu, Y., and He, L. (2011). IPF: In-place x-filling to mitigate soft errors in sram-based FPGAs. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 482 –485.

Freiman, C. (1962). Optimal error detection codes for completely asymmetric binary channels. *Information and Control*, **5**(1), 64 – 71.

Fuchs, W., Chen, C.-Y., and Abraham, J. (1987). Concurrent error detection in highly structured logic arrays. *Solid-State Circuits, IEEE Journal of*, **22**(4), 583 – 594.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

Gorshe, S. and Bose, B. (1996). A self-checking ALU design with efficient codes. In *Proceedings of 14th VLSI Test Symposium, 1996*, pages 157 –161.

Grumberg, O., Schuster, A., and Yadgar, A. (2004). Memory efficient all-solutions SAT solver and its application for reachability analysis. In A. Hu and A. Martin, editors, *Formal Methods in Computer-Aided Design*, volume 3312 of *Lecture Notes in Computer Science*, pages 275–289. Springer Berlin / Heidelberg.

Guzman-Miranda, H., Sterpone, L., Violante, M., Aguirre, M., and Gutierrez-Rizo, M. (2011). Coping with the obsolescence of safety- or mission-critical embedded systems using FPGAs. *Industrial Electronics, IEEE Transactions on*, **58**(3), 814 –821.

Hu, Y., Feng, Z., He, L., and Majumdar, R. (2008). Robust FPGA resynthesis based on fault-tolerant boolean matching. In *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, pages 706 –713.

Jha, N. and Wang, S.-J. (1991). Design and synthesis of self-checking vlsi circuits and

systems. In *Computer Design: VLSI in Computers and Processors, 1991. ICCD '91. Proceedings, 1991 IEEE International Conference on*, pages 578 –581.

Jing, N., Lee, J.-Y., He, W., Mao, Z., and He, L. (2011). Mitigating FPGA interconnect soft errors by in-place lut inversion. In *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, pages 582 –586.

Kang, H.-J. and Park, I.-C. (2003). SAT-based unbounded symbolic model checking. In *Design Automation Conference, 2003. Proceedings*, pages 840 – 843.

Knight, J. C. (2002). Safety critical systems: Challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 547–550, New York, NY, USA. ACM.

Krasniewski, A. (2006). Low-cost concurrent error detection for fsms implemented using embedded memory blocks of FPGAs. In *Design and Diagnostics of Electronic Circuits and systems, 2006 IEEE*, pages 178 –183.

Lo, J.-C., Thanawastien, S., and Rao, T. (1989). Concurrent error detection in arithmetic and logical operations using berger codes. In *Computer Arithmetic, 1989., Proceedings of 9th Symposium on*, pages 233 –240.

Lonsing, F. and Biere, A. (2010). DepQBF: A dependency-aware QBF solver. *JSAT*, **7**(2-3), 71–76.

Lyons, R. E. and Vanderkulk, W. (1962). The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, **6**(2), 200 –209.

Mitra, S. and McCluskey, E. (2000). Which concurrent error detection scheme to choose? In *Test Conference, 2000. Proceedings. International*, pages 985 –994.

Normand, E. (1996). Single event upset at ground level. *Nuclear Science, IEEE Transactions on*, **43**(6), 2742 –2750.

Pradhan, D. K., editor (1986). *Fault-tolerant computing: theory and techniques*, volume 1. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Reddy, E., Sashikanth, V., and Kamakoti, V. (2005). Cluster-based detection of SEU-caused errors in LUTs of SRAM-based FPGAs. In *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, volume 2, pages 1200 – 1203.

Smith, J. and Metze, G. (1978). Strongly fault secure logic networks. *Computers, IEEE Transactions on*, **C-27**(6), 491 –499.

Taber, A. and Normand, E. (1993). Single event upset in avionics. *Nuclear Science, IEEE Transactions on*, **40**(2), 120 –126.

Touba, N. and McCluskey, E. (1997). Logic synthesis of multilevel circuits with concurrent error detection. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, **16**(7), 783 –789.

Wadsack, R. L. (1978). Fault modeling and logic simulation of CMOS and MOS integrated circuits. *AT T Technical Journal*, **57**, 1449–1474.

Wassyng, A., Lawford, M., and Hu, X. (2005). Timing tolerances in safety-critical software. In *Proceedings of the 2005 international conference on Formal Methods*, FM'05, pages 157–172, Berlin, Heidelberg. Springer-Verlag.

Yang, S. (1991). Logic synthesis and optimization benchmarks user guide version 3.0.