Automated Software Solutions to Logic Restructuring and Silicon Debug

by

Yu-Shen Yang

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2010 by Yu-Shen Yang
Abstract

Automated Software Solutions to Logic Restructuring
and Silicon Debug

Yu-Shen Yang
Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto
2010

With the growing size of modern integrated circuit designs, automated design tools have taken an important role in the development flow. Through the use of these tools, designers can develop circuits in a robust and systematic manner. However, due to the high complexity of the designs and strict resource constraints, it is inevitable that mistakes will be made during the design process. Today, a significant amount of development time is dedicated to pre- and post-silicon verification and debug, which can increase the production cost and jeopardize the future growth of the industry. Hence, there is an urgent need for scalable automated verification and debugging techniques, as well as new methodologies that improve circuits in order to reduce errors.

This dissertation presents a set of methodologies to automate three important processes in the VLSI design flow that are related to improving the quality of designs.

The first contribution, automated logic restructuring, is a systematic methodology used to devise transformations in logic designs. This technique can be used for a wide range of post-synthesis applications, such as logic optimization, debugging and engineer change orders, which modify synthesized designs to accommodate their goals. Better results can be achieved if there are various transformations for those applications to select. Experiments demonstrate that the proposed technique is capable of re-structuring designs at a location where other methods fail and also identifies multiple transformations for each location.

The second contribution is a logic-level, technology-independent soft error rate mitigation technique. Soft errors are transient logic pulses induced by radiation from the environment
or released from the silicon chip package materials. This technique identifies conditions where soft errors can cause discrepancies at the primary outputs of the design, and eliminates those conditions through wire replacement. Experimental results confirm the effectiveness of the proposed technique and show that the soft error rate can be reduced at no or small additional overhead to other design parameters.

The final contribution of the dissertation is a software environment for post-silicon debug. The proposed algorithms analyze the data collected during operating silicon chips at speed in the test system and provide useful information to expedite the debugging process. Experiments show that the proposed techniques eliminate one third of design modules, which are suspected as the root cause of the failure. Such a reduction can save engineers time on manual inspection and shorten the turnaround time of silicon debug.
Acknowledgements

I would like to express my gratitude to my supervisor, Prof. Andreas Veneris, for his encouragement and support during my period of study. His guidance and advice are essential to the success of this work.

Many thanks to my PhD committee members, Prof. Jason Anderson, Prof. Stephan Brown, Prof. Jianwen Zhu and Prof. Masahiro Fujita for their through reviews of my dissertation and suggestions for improving this dissertation.

I would also like to thank Prof. Nicola Nicolici from McMaster University, Dr. Subarna Sinha from Synopsys Inc. and Prof. Robert K. Brayton from University of California, Berkeley for their comments and assistance, which are valuable to the success.

I would like acknowledge my appreciation of the help, both academically and socially, from my colleagues at the University of Toronto and Venna Technologies Inc. Special thanks to Sean Sarfarpour, Brian Keng, Yibin Chen, Hratch Mangassarian, Duncan Smith, Alan Baker, Patrick Haline and Evean Qin.

Finally, I would like to thank my parents for their constant support and encouragement. Their hard work and sacriﬁce provide me with a comfortable environment to concentrate on my work.
# Contents

List of Tables ix

List of Figures xi

1 Introduction 1

1.1 Design Optimization and Logic Restructuring ........................................... 5
1.2 Design Reliability ......................................................................................... 7
1.3 Post-silicon Debug ....................................................................................... 8
1.4 Motivation and Contribution ......................................................................... 9
  1.4.1 Automated Logic Restructuring .............................................................. 10
  1.4.2 Soft Error Rate Minimization ................................................................. 11
  1.4.3 Software Solutions to Silicon Debug .................................................... 12
1.5 Dissertation Outline .................................................................................... 13

2 Background 15

2.1 Boolean Satisfiability .................................................................................. 15
  2.1.1 Satisfiability in CAD for VLSI ............................................................... 16
  2.1.2 UNSAT Cores ....................................................................................... 17
2.2 Abstract Representations of Circuits .......................................................... 19
  2.2.1 Modelling Circuits with Binary Decision Diagrams ............................. 21
  2.2.2 Modelling Circuits in Boolean Satisfiability ......................................... 22
2.3 Sets of Pairs of Functions to be Distinguished ............................................ 23
  2.3.1 Formal Definition of SPFDs ................................................................. 25
2.3.2 Graphical Representation of SPFDs ........................................... 28
2.3.3 Computing SPFDs ................................................................. 30
2.3.4 Properties of SPFDs .............................................................. 34

2.4 Debugging Methodologies ......................................................... 36
2.4.1 SAT-based Debug Methodology .............................................. 37
2.4.2 Hierarchical Diagnosis ......................................................... 39

3 Automated Logic Restructuring with ωSPFDs .................................. 41
3.1 Introduction ................................................................................. 41
3.2 Preliminaries .............................................................................. 44
    3.2.1 Dictionary-Based Correction .............................................. 46
    3.2.2 Redundancy Addition and Removal .................................... 47
    3.2.3 Diagnosis-Based Rewiring .................................................. 50
    3.2.4 Logic Rewiring for Sequential Circuits .............................. 51
    3.2.5 SPFD-Based Optimization .................................................. 51
3.3 Motivation .................................................................................. 52
    3.3.1 Error Equation .................................................................. 52
    3.3.2 Limitations of the Dictionary Model ................................... 54
3.4 Approximating SPFDs ............................................................... 56
    3.4.1 Computing ωSPFDs for Combinational Circuits .................. 58
    3.4.2 Computing ωSPFDs for Sequential Circuits ...................... 61
    3.4.3 Optimizing ωSPFDs with Don’t Cares ................................. 64
    3.4.4 Validating ωSPFDs ............................................................. 68
3.5 Logic Transformations with ωSPFDs ............................................. 68
    3.5.1 SAT-based Searching Algorithm ....................................... 71
    3.5.2 Greedy Searching Algorithm .............................................. 74
    3.5.3 On the Quality of Test Vector Simulation for SPFD Approximation ... 75
3.6 Extension to Multiple Locations .................................................. 78
3.7 Experimental Results ............................................................... 80
List of Tables

2.1 SAT formula for simple gates ........................................ 22
2.2 Truth table of functions that can distinguish the pair (ab, \overline{ab}) ........................................ 26
2.3 Truth table of functions that can distinguish the pair (\overline{a}b, ab) ........................................ 27

3.1 Common design error types ........................................ 47
3.2 Quantifying logic transformations ........................................ 55
3.3 Characteristics of benchmarks ........................................ 81
3.4 Combinational logic transformation results ........................................ 82
3.5 Sequential logic transformation results ........................................ 88
3.6 Logic transformation at multiple locations ........................................ 91

4.1 Characteristics of benchmarks ........................................ 105
4.2 Experimental results using the cost function (OnlySER) ........................................ 106
4.3 Experimental results using the cost function (SER\&All) ........................................ 107

5.1 Operation of primitives with X ........................................ 128
5.2 Three-value logic encoding ........................................ 135
5.3 Three-value SAT formula for simple gates ........................................ 136
5.4 Performance with limited hardware constraints ........................................ 143
5.5 Traceable register group information ........................................ 145
5.6 Performance of debugging with proposed techniques ........................................ 146
5.7 Performance of the search algorithm ........................................ 148
5.8 Impact of state information on the diagnosis ........................................ 149
5.9 Impact of window sizes on the searching algorithm . . . . . . . . . . . . . . . . . 149
List of Figures

1.1 The four stages of the integrated circuit design flow ............................................. 2
1.2 Level of abstraction of design modelling .............................................................. 3

2.1 Representations of a sequential circuit ................................................................. 20
2.2 An example circuit and its BDD representation ....................................................... 21
2.3 Example of Boolean relation .................................................................................. 24
2.4 The graphical representation of an SPFD ............................................................... 29
2.5 Example of computing SPFDs .................................................................................. 31
2.6 Specialized miter for computing SPFDs [1] ............................................................ 33
2.7 SPFDs of a two-input OR-gate ............................................................................... 34
2.8 SAT-based diagnosis construction for sequential circuits ...................................... 39
2.9 Hierarchical diagnosis ............................................................................................ 40

3.1 Miter structures to identify the required function at η in [2] ................................... 45
3.2 Example of a circuit with redundancy connections .................................................. 48
3.3 Optimization with RAR ........................................................................................ 49
3.4 Circuitry of the error equation ................................................................................ 53
3.5 The circuit for Examples 3.3 and 3.7 ..................................................................... 59
3.6 SPFD and aSPFDs of the nodes in Figure 3.5 .......................................................... 59
3.7 The circuit for Example 3.4 .................................................................................... 63
3.8 State transition ........................................................................................................ 67
3.9 aSPFDs of nodes in Figure 3.5 ................................................................................. 70
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.10</td>
<td>Example circuit in Figure 3.5(a) after logic transformation</td>
<td>70</td>
</tr>
<tr>
<td>3.11</td>
<td>Example of a SPFD and a local SPFD</td>
<td>75</td>
</tr>
<tr>
<td>3.12</td>
<td>Relation of two transformation locations</td>
<td>78</td>
</tr>
<tr>
<td>3.13</td>
<td>Additional wire selection effort</td>
<td>84</td>
</tr>
<tr>
<td>3.14</td>
<td>Performance of restructuring with variable vector sizes for combinational designs</td>
<td>85</td>
</tr>
<tr>
<td>3.15</td>
<td>Performance of restructuring with variable vector sizes for sequential designs</td>
<td>89</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of minimum SPFDs</td>
<td>99</td>
</tr>
<tr>
<td>4.2</td>
<td>Example circuit in Figure 4.1 and its minimum SPFDs after rewiring</td>
<td>102</td>
</tr>
<tr>
<td>5.1</td>
<td>Typical silicon debug flow</td>
<td>111</td>
</tr>
<tr>
<td>5.2</td>
<td>Common Design-for-Debug techniques</td>
<td>113</td>
</tr>
<tr>
<td>5.3</td>
<td>A single debug analysis session</td>
<td>120</td>
</tr>
<tr>
<td>5.4</td>
<td>Hierarchical diagnosis</td>
<td>121</td>
</tr>
<tr>
<td>5.5</td>
<td>Timeframe modules modelled with multiplexers</td>
<td>123</td>
</tr>
<tr>
<td>5.6</td>
<td>Timeframe diagnosis</td>
<td>124</td>
</tr>
<tr>
<td>5.7</td>
<td>Timeframe diagnosis with multiple error excitation</td>
<td>125</td>
</tr>
<tr>
<td>5.8</td>
<td>X-simulation</td>
<td>128</td>
</tr>
<tr>
<td>5.9</td>
<td>Circuit for Example 5.2</td>
<td>130</td>
</tr>
<tr>
<td>5.10</td>
<td>Test vector trace reduction</td>
<td>133</td>
</tr>
<tr>
<td>5.11</td>
<td>An unfolded circuit from $T_{k-w}$ to $T_{k+w}$</td>
<td>134</td>
</tr>
<tr>
<td>5.12</td>
<td>The model of target registers</td>
<td>138</td>
</tr>
<tr>
<td>5.13</td>
<td>The model of candidate registers</td>
<td>138</td>
</tr>
<tr>
<td>5.14</td>
<td>ILA of the example circuit in Figure 5.9(a)</td>
<td>139</td>
</tr>
<tr>
<td>5.15</td>
<td>Impact of depth $n$ on size of suspects</td>
<td>144</td>
</tr>
<tr>
<td>5.16</td>
<td>Impact of interval size $k$ on width of critical interval</td>
<td>144</td>
</tr>
<tr>
<td>5.17</td>
<td>Performance of the search algorithm</td>
<td>150</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Since the early 1960s when the first integrated circuits (ICs) were fabricated, the technology for developing ICs has dramatically evolved. Starting from a simple circuit with only a handful of transistors, the number of transistors that can be put on a single silicon chip has been doubling approximately every two years as predicted by Moore’s Law [3]. With the current 45nm technology, we have reached integration levels exceeding one billion transistors per silicon chip. This high level of integration makes it possible to develop complex circuits that can perform a multitude of functions. As a result, developing such complex IC designs has become a challenging task. The situation gets more intense when development is restricted by time. In order to fulfill the high demands of the consumer, electronic products need to be delivered to the market within tighter time-to-market constraints. Without automation, manually designing chips can be impractical and error-prone.

The semiconductor industry has products that reach all aspects of commercial and consumer markets, domestically and internationally. This industry consistently creates smaller, faster and more powerful chips which fuel the accelerated demand for the end products. Chip companies are challenged daily to design increasingly complex devices with increased functionality, while remaining cost competitive. The rapid growth of the industry in the past 30 years is in part attributed to advances in the Electronic Design Automation (EDA) community that develop Computer-Aided Design (CAD) tools to aid the human engineer in designing such complex high-performance devices. In the past few decades, various processes in the design flow for
Very Large Scale Integration (VLSI) systems have been automated with CAD tools, in order to satisfy given performance and design time constraints. Those tools make the design process more efficient and robust, while increasing the reliability of the design.

Broadly speaking, creating integrated circuits involves four stages as shown in Figure 1.1: design, fabrication, test, and packaging [4]. During the design stage, the idea for a design is cast into a series of models at different levels of abstraction. Due to the complexity of IC designs, it makes sense to develop the design in a top-down fashion; the design begins with a high-level abstracted description and more functional details are progressively added to the model. Typical abstraction levels used in the development of digital designs are depicted in Figure 1.2 [4–6].

1. A functional specification provides the behavioral view of a design. It describes the function of the circuit regardless of its implementation. As the specification of a design is often documented in a natural language, a functional specification realizes the specification as an executable program using a high-level algorithm expressed in a programming language, such as Matlab, C, C++. Such a model is used to verify that the system performs as required.

2. A structural description presents the block-level structure of the design. It is often modelled using Register Transfer Level (RTL) structures with a Hardware Description Lan-
High level languages (C, C++)

Functional Specification
High level languages (C, C++)

Behavioral Synthesis

RTL (VHDL, Verilog)

Structural Description
RTL (VHDL, Verilog)

Logic Synthesis

CMOS logic netlist

Logic Description
CMOS logic netlist

Physical Design

GDSII

Physical Layout
GDSII

Silicon

Figure 1.2: Level of abstraction of design modelling

guage (HDL) such as Verilog or VHDL. A structure description includes a list of functional modules and specifies how modules are interconnected to perform the behavior described in functional specifications. Information about a design such as hierarchy, registers, or time behavior is also included in this model.

3. A logic description provides gate-level details of a system. At this level, cell libraries for the targeted fabrication technology are used to model the system in terms of Boolean logic gates, latches, and flip-flops.

4. Physical layout consists of multiple metal layers in correspondence with the masks used for chip fabrication. Parasitic resistance and capacitance can be extracted from the layout to aid in the accuracy of circuit simulation.

Models with different abstraction information allow designers to pick a model that best fits to tasks performed without being disturbed by unnecessary details. The process by which the model is refined is referred to as synthesis. Depending on input models, different synthesis procedures are carried out as shown in Figure 1.2. The ultimate goal of circuit synthesis is to generate a detailed model of a design that contains all required features for fabricating chips.
Synthesis processes are often accompanied by *circuit optimization*, which enhances the overall quality of the design. The quality of the design is evaluated by several design parameters, such as area, power, delay, etc.

*Verification* is another important step in the design stage. Clearly, an error in any model can put the success of the entire product in jeopardy. If errors are not detected early enough in the design flow, they can eventually reside in the silicon. Hence, it is necessary to validate the correctness of each model after synthesis. Verification procedures check the consistency between models and ensure that properties of the design are not violated. Verification can be performed through *simulation-based* methods or *formal verification* methods. Of these two approaches, simulation-based methods are widely used techniques for verifying a design [7], and are based on the simulation results of designs under given sets of input patterns. Consequently, the quality of verification results depends on given simulation patterns. Errors can be missed if simulation patterns are not selected carefully to exercise the design. Simulation lacks of robustness and is a non-exhaustive method. Nevertheless, due to its simplicity, simulation is still widely adopted by the industry for verification.

In contrast, formal verification methods use mathematical techniques to exhaustively check all possible input patterns to the design and to detect their consistency [7]. Since the complete input space is explored, the result of the validation is guaranteed. Although attractive, the drawback of formal verification methods is scalability as they are usually limited to designs with a few million of gates.

Once the design is complete and verified, it is provided to an IC fabrication plant in a format such as GDSII to be implemented on silicon. Before the design goes into production, silicon prototypes are first fabricated and tested. *Post-silicon validation and debug* verifies the functionality of the silicon prototype against the functional specification. At this stage, any design errors missed by the verification processes in the design stage are captured. To differentiate this process from verification during the design stage, the latter is often referred to as *pre-silicon verification*.

*Manufacture test* is another testing process carried out on fabricated chips. The complex manufacturing process is not perfect and physical defects, such as metal-to-metal shorts or
discontinuous wires, can be introduced. Manufacture test captures those defects and separates malfunctioning circuits from good ones before they are sliced and packaged into silicon chips.

During verification, if the design or silicon prototype is shown to be faulty, it undergoes design debugging to discover the source of errors. It has been reported in [8] that verification and debugging today take as much as 70% of the overall chip design time, with this time equally split between these two tasks.

In this four-stage flow, the human expertise is complemented by many CAD tools. Before any CAD tools were developed, integrated circuits were designed by hand and manually laid out. Today, CAD tools fully or partially automate the manual steps. Most notably, design synthesis, place and route, and automated test pattern generation (ATPG) tools populate the fast growing EDA industry [9].

Although the development of automated tools for many steps in the flow has matured, there are some areas where automation is still lacking or needs improvement. In particular, logic restructuring in the design stage and silicon debug after the prototype fails post-silicon validation still lack significant automation. The purpose of this dissertation is to address those concerns. In the following sub-sections, such problems are examined in more detail.

1.1 Design Optimization and Logic Restructuring

During the logic synthesis phase of the design flow, the design is optimized for various objectives to maximize circuit quality. Common quality measures for optimization are area, delay, and power consumption [10–12]. In addition, there is extensive work on optimizing designs for better testability and reliability [13, 14].

The optimization process is divided into two steps: technology-independent optimization and technology-dependent optimization. The goal of technology-independent optimization is to generate an optimum abstract representation of the design without being aware of the underlying technology that is used to implement the design. In the context of this dissertation, optimum means a design with improved performance with respect to the amount of resources dedicated by the designer to do so. Symbolic methods use Boolean and algebraic techniques,
such as node factoring and substitution, to simplify the equations that represent the synthesized networks [15, 16]. In the second step, technology-dependent optimization, the logic netlist is further optimized according to the given logic library. The key to this step is to fully utilize components in the library such that the resulting netlist complies with its area, speed and testability constraints.

The global optimization steps outlined above are usually followed by a more careful local optimization step targeted to achieve a specific performance goal. This is usually carried out through an iterative sequence of logic restructuring, such as design rewiring operations [17, 18]. During each iteration of this process, a target wire is identified to be removed due to the violation of some constraints (for example, a wire may be on a critical path, or it has excessive power consumption). Those rewiring techniques identify an alternative connection in the design such that the overall functionality of the design remains unaffected. Optimization not only changes the combinational structure, but also alters the memory elements. Retiming [10] and Sequential Redundancy Addition and Removal [19] are examples of the optimization techniques that minimize the number of memory elements and relocate them so that the circuit can operate with a faster clock.

After logic optimization, the gate-level netlist is verified to ensure its functional correctness. If verification fails, the source of the error must be identified and rectified before proceeding further. In this case, direct rectifying a gate-level netlist, instead of the RTL representation, is preferred to avoid repeating the time-consuming synthesis and optimization processes. Constructing corrections manually requires designers to understand the gate-level netlist. This can be difficult for designers who are usually only familiar with RTL representations. A mapping between variables in the RTL representation and nodes in the synthesized gate-level netlist may not exist after logic synthesis and optimization. Furthermore, because the process is automated, the names of nodes in the gate-level netlist may not be easily recognized. Usually, only key variable names in the RTL representation are retained. As a result, it is hard for designers to construct rectifications at gate-level manually. Tools that can analyze designs and automate the process of logic restructuring are therefore necessary.

Another situation where logic restructuring is required is in the Engineering Change (EC)
process. In a typical IC development cycle, it is common to change the specification during the development to meet new constraints or criteria, even though the design has been completed or partially completed. These last-minute design changes are commonly referred to as *Engineering Change Orders* (ECOs). Since a lot of time and engineering effort have been invested in synthesizing and optimizing the design, it is preferable to modify the design directly on the gate-level netlist. Again, due to the information loss during the transition from the RTL to the post-synthesis representation, it is not obvious to designers how to modify the design without the help of software analysis.

### 1.2 Design Reliability

Another important task during the development of IC designs is to estimate and improve reliability. Conventionally, the reliability of a system is defined as the probability that the system will perform its required function for a specific period of time under stated conditions [20]. In the last few decades, the development of IC designs was largely driven by achieving better performance, lower cost, and efficient power consumption. As the system complexity increases, there is a higher demand for more reliable IC products, such as devices for telecommunication and satellites; it is absolutely essential that those devices do not fail. Hence, semiconductor reliability has become one of the major concerns in semiconductor manufacturing and needs to be addressed at the pre-product stage [21].

In order to improve reliability, the concept of *Design-for-Reliability* (DfR) has become a part of the technology development process. Designers need to understand the physics-of-failures in order to deal with these problems. The failure of semiconductor devices can occur due to various physical mechanisms. Some mechanisms, such as electromigration and hot carriers, can cause permanent damage to devices, while others, such as soft errors due to radiation, only affect the devices temporarily. Many researchers have focused on preventing these failure mechanisms from occurring by altering physical-level designs, improving the fabrication process and utilizing materials with different characteristics [21].

In addition to hardening designs at the lower level, several techniques to enhance designs
in their logic-level representation [22–24] are proposed. Such approaches make it possible to consider reliability as a design objective much earlier in the design cycle. As a result, other design parameters can be considered collectively to obtain a balanced optimization.

1.3 Post-silicon Debug

Post-silicon debug, or silicon debug for short, is a process that still lacks automated and robust debugging strategies to efficiently identify the root-cause of failures in silicon prototypes. The goal of silicon debug is to capture any functional errors or defects in the silicon prototype.

A defect is a physical imperfection in silicon chips. It occurs either during the manufacture or use of devices. A representation of a defect in an abstracted design model is referred to as a fault. For example, a metal wire may be accidentally connected to the ground during the manufacture. Such a defect is called a short to ground. The fault for this defect is stuck-at logic 0. Simply put, defects and faults are the imperfections in the hardware and function, respectively.

Traditionally, functional errors are captured early in the pre-silicon verification using formal verification or simulation techniques. As discussed earlier, formal verification techniques mathematically prove that designs meet the requirements and that undesired behavior does not occur. Formal verification techniques are the most powerful because they explore the complete input space of the design under verification. However, since the formal proof process can be resource-intensive, those techniques are usually limited to verifying small-sized designs [25]. Simulation techniques are the primary tool used for functional verification of large designs. Engineers simulate the design with various input vectors to excite different functionality. However, it is difficult to determine whether all corner cases are considered. Furthermore, simulating a chip operating for one minute in the actual clock frequency in the system could take a week to complete. Consequently, it is too time intensive to test all possible inputs and states of a system to capture all functional errors.

Due to the growing complexity of integrated circuits, the time and resources required to thoroughly verify the functionality of a circuit using pre-silicon verification techniques has become
unacceptably large. As a result, functional errors may escape from pre-silicon verification [26].

It has been reported that more than half of the failures found in the design tape-outs are due to logical or functional errors not discovered during pre-silicon verification [27]. In addition, as the feature size of the semiconductor decreases, a small deviation during the fabrication can have a greater impact on the performance of silicon chips. Such process variation is not modelled accurately in the pre-silicon stage. As a result, the silicon chips may behave differently than what is expected from the simulation. In order to expose the aforementioned unexpected behavior, silicon chips are operated at speed in a real-world working environment during silicon debug. This makes it possible to test the design with longer vector sequences that cannot be done in a software simulation.

Currently, the success of silicon debug highly depends on the engineer’s experience and knowledge of the design. Engineers use various hardware test tools to understand the behavior of the faulty chip and determine the root cause of the failure. Hence, silicon debug can be a tedious and time-consuming process. Nevertheless, it is critical to have a silicon debug process that can pinpoint the root cause with a short turnaround time, so that designers can quickly respond and rectify the design. To fix functional errors, the design needs to undergo another round of the design cycle, starting from re-synthesis to correct its functionality. The development cycle can easily be prolonged. This results in postponement of the delivery of the final product to the market, an event that can have catastrophic budget consequences to the manufacturer.

More importantly, each re-spin implies a increase in the cost of product development which lowers the profit margin. Therefore, the need for a software environment that can automate the debugging analysis and provide an accurate root cause analysis is a necessity to maintain the quality of the product while meeting the original time constraints.

1.4 Motivation and Contribution

In 2006, International Technology Roadmap for Semiconductors (ITRS) issued its new set of needs for the current and next generation design processes for semiconductors [28]. In that
Chapter 1. Introduction

Edition, the roadmap contains a major update in design verification and validation. The report states that “technological progress depends on the development of rigorous and efficient methods to achieve high-quality verification results—-and techniques to ease the burden of debugging a design once a bug is found”. It continues, “Verification must increasingly integrate more formal and semi-formal engines to complement simulation—-Without major breakthroughs, verification will be a non-scalable, show-stopping barrier to further semiconductor progress.” Without a doubt, the roadmap depicts a grim yet realistic picture that establishes an urgent need for scalable automated verification and debugging techniques and new methodologies that improve circuits to reduce errors.

The compelling tone in the ITRS report and recent data from silicon test failures reflect the widening divergence between current chip design practices and the state-of-the-art practices in EDA verification, design debug and silicon debug. This dissertation focuses on automating processes in three important topics in the VLSI design flow that help to alleviate the pain described by ITRS and also allow for robust new design methodologies. The contributions of the thesis are summarized as follows:

- An algorithmic methodology to restructure logic-level netlists, the results of which allow for more flexibility during rewiring-based optimization.
- A novel logic optimization methodology to mitigate the impact of soft errors on designs.
- A software environment for post-silicon debug to identify the root cause of failures in a silicon prototype.

The contributions in each topic are discussed in detail in the following subsections.

1.4.1 Automated Logic Restructuring

Common logic restructuring techniques usually limit the restructuring to a set of simple models that consist of the addition or removal of a single wire or gate [29]. Later this dissertation shows that such a simple model is not enough to explore the complete functional flexibility of the design. To explore other potential solutions, Sets of Pairs of Functions to be Distinguished

Sets of Pairs of Functions to be Distinguished
(SPFDs), which are a different representation of the functional flexibility of a design, are utilized.

In summary, this work makes the following contributions:

- It presents a simulation-based technique to approximate SPFDs (referred to as $\alpha$SPFDs) in order to alleviate the runtime and memory issues that formal SPFD generation approaches may encounter. The technique can handle both combinational and sequential designs.

- An algorithm to construct logic restructuring is proposed. The algorithm utilizes $\alpha$SPFDs as the guideline to construct the required transformation to make the new design comply with the desired specification. The generated transformation can be applied for various problems, such as logic optimization, engineering changes (EC), etc.

- Two approaches are presented to construct logic transformations: a SAT-based approach and a greedy approach. The SAT-based approach models problems as a Boolean satisfiability instance and is able to find the optimal solution, while the greedy approach returns a non-optimal solution with faster computation.

- The extension of the transformation algorithm to deal with multiple locations is also presented. Applying transformations at multiple locations simultaneously is a practical issue. For instance, in the concept of logic correction, if an error is located in the synthesis tool, it can be instantiated at multiple locations in the design. In this case, a rectification algorithm that performs corrections at multiple locations in the logic network is necessary.

With the algorithmic construction of alternative logic structures, designers have more candidates to select from to best fit the desired goal and obtain a better solution.

1.4.2 Soft Error Rate Minimization

Traditionally, design optimization during the synthesis phase focuses on improving the area and performance of designs. In recent years, there has been much research focused on optimizing designs for testability and reliability, and in particular, on strengthening designs to protect against soft errors. Soft errors are induced by radiation in the operating environment or released from the chip packaging materials. Radiation generates electron-hole pairs in the silicon and
flips the logic value when sufficient amounts of electrons are collected. The rate at which soft
errors occur is referred to as soft error rate (SER). Due to the continued trend in technology
scaling (e.g., smaller feature size, lower supply voltages, etc.), digital designs are becoming more
vulnerable to soft errors. Since the logic values of a node can easily be flipped with only a small
amount of radiation, it is important to ensure that the design is soft error tolerant.

The flip, called single-event transient (SET), caused by soft errors can only be effective if it
is propagated to the primary outputs or latched by register elements. Therefore, the technique
presented in the dissertation reduces SER by restructuring the logic design to eliminate the
condition for the observation of SETs.

1.4.3 Software Solutions to Silicon Debug

Both RTL debugging and silicon debug identify the source of errors in the design against the
functional specification. However, there are differences between these two procedures. First, the
observability of internal signals in a silicon prototype is restricted in both space and time. While
values of any internal signals in a RTL design can be obtained via simulation, signals in a silicon
prototype can only be observed if they are connected to an output port, or accessed through
hardware components, called Design-for-Debug. In spite of the fact that Design-for-Debug
techniques provide access to internal signals, algorithms that analyze acquired data to help the
post-silicon debugging process are lacking. Second, because of the limited amount of signal
data acquired from the prototype, it usually takes several debug iterations to fully understand
the behavior of the error and identify the root cause. Third, during silicon validation, the
silicon prototype is operated at-speed in the system that it is designed for. The test trace for
debugging can be orders of magnitude longer compared to the trace used in RTL debugging.
It is important to identify the segment of the trace that really matters in order to simplify the
analysis.

With the aforementioned observation, a software environment that provides the following
solutions to improve the efficiency and accuracy of silicon debug is presented:

- It introduces an automated software-based debug methodology that is complementary to
current data acquisition hardware solutions in silicon chips. The methodology analyzes
the data acquired during the silicon test and helps engineers determine the root cause of the failures.

- It presents a novel timeframe diagnosis that can identify the time interval where the error or fault might have occurred, as it is useful to understand the behavior of the error or fault. In addition, the debug analysis can be constrained to focus on a smaller interval of the erroneous trace, which can speed up the debug analysis.

- A tracing algorithm that utilizes unsatisfied cores (UNSAT cores) to identify gates on error propagation paths is discussed. The common tracing algorithms such as path-trace [30] and X-simulation [31], are pessimistic when determining the paths that the error might travel on. Consequently, the result is too large to be useful. With UNSAT cores, the proposed algorithm can provide results that are more concise and more useful in pruning false suspects.

- A novel SAT-based search algorithm that identifies a potential implication to derive the value of a signal in a design. This algorithm is helpful in alleviating the data acquisition hardware constraint, where only a small set of internal signals can be traced by DfD hardware. Instead of tracing signals of interest directly, they can be obtained by implications from a different set of signals.

Silicon debug is a difficult and complicated problem which lacks of automated software tools. This dissertation shows the potential in automating the data analysis of the silicon debug flow. With a set of various algorithms that prune suspect modules in both spacial and temporal domains, the diagnostic result can be concise and accurate.

1.5 Dissertation Outline

This dissertation is organized as follows. Chapter 2 contains background information that will be used in the rest of the dissertation. This material includes the definition and formulation of Boolean satisfiability and a short description of the concept behind Sets of Pairs of Functions to be Distinguished (SPFD). They are used to model and provide theories to the solutions
presented. The next three chapters describe the work on the following three topics respectively: Chapter 3 discusses the technique to approximate SPFDs, as well as the methodology of logic transformation with αSPFDs; Chapter 4 presents the SER minimization technique using αSPFDs; Chapter 5 describes software solutions to silicon debug. Finally, the conclusion and possible topics for future work are covered in Chapter 6.
Chapter 2

Background

This chapter discusses the basic definitions and concepts that are essential for describing the work presented in this dissertation. Section 2.1 summarizes the concept of Boolean satisfiability, which is used to model designs and to formulate problems presented in this dissertation. Section 2.2 reviews how combinational and sequential designs can be modelled using Binary Decision Diagrams (BDDs) and Boolean satisfiability. Finally, Section 2.3 discusses a different approach for representing design functionality using SPFDs. This provides the theoretical background for the methodologies presented in Chapter 3 and Chapter 4.

2.1 Boolean Satisfiability

The Boolean satisfiability (SAT) problem is a well-known constraint problem that has been applied to many areas in the fields of VLSI computer-aid designs. The input of SAT is a Boolean formula, which is composed of (1) Boolean variables whose values are 0 or 1 (false or true), (2) operators such as \( \neg \) (NOT), \( \cdot \) (AND) and \( + \) (OR), and (3) parentheses. Given a Boolean formula \( \phi \), SAT asks to determine a variable assignment such that \( \phi \) is evaluated to be true. An assignment to \( \phi \) is a set of values to all variables of \( \phi \). When a Boolean variable is assigned with a logic value 1 (0), the variable is said to be set (unset), or activated (deactivated), or selected (unselected). Those terms are used interchangeably throughout the remainder of the dissertation. If there is a solution to the SAT problem, the assignment is called a satisfying
assignment and $\phi$ is said to be satisfiable; otherwise, $\phi$ is unsatisfiable.

In practice, Boolean formulae are usually presented in Conjunctive Normal Form (CNF). A Boolean formula in CNF consists of a conjunction (product) of clauses, where each clause is a disjunction (sum) of literals. A literal is an occurrence of a variable or its negation. To satisfy a Boolean formula, all clauses have to be satisfied. Equivalently, at least one literal in each clause is evaluated to be a logic 1.

**Example 2.1** Below is a Boolean formula with five clauses and four variables.

$$\phi = (a + \overline{c}) \cdot (b + \overline{c}) \cdot (\overline{a} + \overline{b} + c) \cdot (c + d) \cdot (\overline{c} + \overline{d}) \quad (2.1)$$

A satisfying assignment of $\phi$ is $< a = 0, b = 0, c = 0, d = 1 >$, since every clause in $\phi$ contains at least one literal that evaluates to a logic 1.

Most modern SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure [32], which searches satisfying assignments using a branch-and-bound approach. The procedure enters a decision loop wherein it selects an unassigned variable and assigns a value to it. When the procedure makes an assignment decision, it also checks for other variable assignments that can be deduced based on the variable assignments done so far. In this step, a conflict can occur, if implications or decisions from different clauses attempt to assign opposite logic values to the same variable. When a conflict occurs, the procedure backtracks to the most recent decision prior to the conflict and assigns a different value. This is called chronological backtracking. The procedure exits the loop when a satisfying assignment is found (satisfiable) or all possible assignments have been tried (unsatisfiable); that is, a conflict is reached at the decision level 0.

2.1.1 Satisfiability in CAD for VLSI

It is illustrative to briefly discuss SAT in the context of complexity theory. SAT holds a special place in this area, as it is the first problem to be classified as NP-Complete [33]. NP-Complete problems are verifiable in polynomial time but there is no known algorithm that can solve them in polynomial time. Additionally, by definition, any problem that is verifiable in polynomial
time is also polynomial-time reducible to any NP-Complete problem. Informally, this implies that if an efficient algorithm is found to solve one particular NP-Complete problem, then all other such problems can be solved efficiently using the same algorithm.

Since SAT is itself NP-Complete, an efficient SAT solver can be utilized to solve other problems that can be reduced to SAT instances. Therefore, there has been an increased effort in the past two decades to develop efficient engines to tackle SAT instances. Modern SAT solvers use efficient data structure manipulation and advanced techniques, such as conflict-driven learning and conflict-driven backtracking [34–36] to solve CNF formulae with tens of thousands of variables and millions of clauses in just a few minutes. Conflict-driven learning adds conflict clauses to the formula to prevent the procedure from reaching the same conflict in the future. Conflict-driven backtracking allows the procedure to backtrack to the closest decision that caused the conflict (non-chronological backtracking). Although advanced SAT solvers are used in this dissertation, familiarity with the underlying concepts is not required and are beyond the context of this dissertation. The reader can refer to [7, 34–36] for more information.

Due to the rapid advances in SAT solvers in the past decade, it has been shown that the use of SAT provides remarkable performance improvements in many VLSI CAD problems. Today, industrial practices in equivalence checking [37], timing analysis [38], path delay test generation [39], bounded model checking [40], two-level logic minimization [41], FPGA routing [42] and test-generation [43] are based on the fact that SAT solvers can address the inherent theoretical complexity of these problems in an efficient and practical manner.

2.1.2 UNSAT Cores

When a CNF formula is unsatisfiable, any subset of CNF clauses that are unsatisfiable by themselves are called an UNSAT core.

**Definition 2.1** Given a SAT formula $\phi$, $UC$ is an UNSAT core for $\phi$ if and only if $UC$ is a formula $\phi_c$ s.t. $\phi_c$ is unsatisfiable and $\phi_c \subseteq \phi$. 

Modern SAT solvers [34–36] can produce UNSAT cores as a byproduct when proving unsatisfiability. Intuitively, an UNSAT core represents a reason why the SAT instance is unsatisfiable. UNSAT cores have been applied in areas of design error debugging [44], functional verification [45], etc. In this dissertation, they are used in the context of silicon debug to identify how error effects are propagated in a design.

**Example 2.2** The SAT formula

\[ \phi = (a + \overline{c}) \cdot (b + \overline{c}) \cdot (\overline{a} + \overline{b} + \overline{c}) \cdot (c + \overline{d}) \cdot (\overline{a} \cdot \overline{b}) \cdot (c) \cdot (\overline{d}) \]  

(2.2)

is unsatisfiable. One of the UNSAT cores is

\[ \text{UNSAT core} = \{(a + \overline{c}), (\overline{a}), (c)\} \]

The clauses \((\overline{a})\) and \((c)\) set \(a = 0\) and \(c = 1\), respectively. These assignments cause \((a + \overline{c})\) to be unsatisfied.

Note that an unsatisfiable SAT instance can have multiple UNSAT cores. Since each UNSAT core can be considered as a cause of unsatisfiability, it is desired to obtain multiple UNSAT cores for some applications. For example, in this dissertation, multiple UNSAT cores are calculated to reveal different potential error propagation paths.

Additional UNSAT cores can be obtained by eliminating an UNSAT core, as described in [46]. The idea is to break the unsatisfied condition by removing the constraints applied to Boolean variables. Iteratively eliminating UNSAT cores can make the SAT instance become satisfiable eventually. Given an UNSAT core \(UC\), each clause \(c = (v_1 + v_2 + \cdots + v_k) \in UC\) is augmented with a distinct relaxation variable \(r\): replacing clause \(c\) with \(c' = (v_1 + v_2 + \cdots + v_k + r)\). The new clause \(c'\) is satisfied (and hence relaxed) when the variable \(r\) is set to 1, regardless of the assignment to variables \(v_1, v_2, \cdots, v_k\). In other words, the modification allows the SAT solver to remove the constraints applied by the original clause \(c\). Since the unsatisfiable condition is broken, \(UC\) is no longer unsatisfiable. To ensure that only one clause of the UNSAT core is relaxed, additional clauses are added to the CNF to enforce an one-hot constraint on the relaxation variables. That is, one and only one of the relaxation variables can be set to true. This is necessary to ensure that the constraints are not over-relaxed. Depending on the application,
users may only want a subset of clauses in the SAT instance to be relaxable. Thus, only certain constraints can be violated. For instance, in [47], the authors apply Boolean satisfiability to perform design debugging. Since the goal is to find the errors in a design, in their formulation, clauses representing the design are relaxable while clauses that constrain the primary inputs and primary outputs are not.

Example 2.3 Continuing from Example 2.2 assume that clauses $(\overline{a}), (c)$ in Equation 2.2 are relaxable. Another UNSAT core can be extracted by relaxing $(\overline{a}), (c)$ with $r_1$ and $r_2$. The new formula becomes

$$\phi' = (a + \overline{a}) \cdot (b + \overline{a}) \cdot (\overline{a} + b + c) \cdot (c + d) \cdot (\overline{c} + \overline{d}) \cdot (\overline{a} + r_1) \cdot (b) \cdot (c + r_2) \cdot (d) \cdot (r_1 + r_2) \cdot (\overline{r_1} + \overline{r_2})$$

(2.3)

where $(r_1 + r_2)$ and $(\overline{r_1} + \overline{r_2})$ represent one-hot encoding which ensures only one relaxable clause is relaxed. In this new formula, the UNSAT core in Example 2.2 is resolved (by assigning 1 to $r_1$ or $r_2$). The new UNSAT core is

$$\text{UNSAT core} = \{(b + \overline{a}), (c + d), (\overline{b}), (\overline{d})\}$$

2.2 Abstract Representations of Circuits

This section outlines two abstract circuit representations, namely BDD and SAT. First, the two types of circuits that will be used in this dissertation are defined as follows.

Definition 2.2 A circuit is combinational if it computes a function which depends only on the values of the primary inputs applied to the circuit. For each input vector, there is only one response at the primary outputs.

Definition 2.3 A circuit is sequential if it computes a function that depends both on the present values of its primary inputs and the values applied to the primary inputs at some previous time.
In this dissertation, a sequential circuit consists of a combinational circuitry that is fed by the primary inputs ($X = \{x_i, \cdots, x_m\}$) and the state variables ($S = \{s_i, \cdots, s_p\}$) as shown in Figure 2.1(a). The values of the state variables depend on the values of the primary inputs in previous cycles and the initial state. As a result, the primary outputs ($O = \{o_1, \cdots, o_n\}$) of the circuit are changed after a series of changes in the primary inputs. When analyzing sequential circuits, they are usually modelled as pseudo-combinational circuits, as shown in Figure 2.1(b). This representation is referred to as the Iterative Logic Array (ILA) representation of the circuit, also known as time frame expansion [48]. In the ILA representation, the design is unfolded over time to maintain its combinational functionality. Each $C^i$ of the array is identical to the combinational circuitry of the original sequential circuit. In this representation, the superscript of a symbol refers to the cycle of the unfolded circuit. For instance, $X^2$ represents the set of the primary inputs in the second cycle. The terms timeframe $i$ and cycle $i$ both represent the $i^{th}$-copy of the combinational circuitry in the ILA representation of the sequential circuit and are denoted as $T_i$. The ILA representation makes it possible to apply methodologies used for combinational designs to sequential ones with minor modifications. The side-effect of the
ILIA representation is that the input vector space of the unrolled circuit grows exponentially in relation to the number of cycles the circuit has unrolled. This can become computationally expensive for some methods if the size of the underlying data structure is correlated with the number of primary inputs [49].

2.2.1 Modelling Circuits with Binary Decision Diagrams

Reduced Ordered Binary Decision Diagrams (ROBDDs) [15], often referred to as BDDs, are the most frequently used data structure for the representation of Boolean functions in the area of VLSI CAD applications. They are a canonical representation of Boolean functions and are more compact compared to other alternatives, such as truth tables, sum-of-products, etc. [7].

A BDD is a directed acyclic graph consisting of decision nodes and terminal nodes. There are exactly two terminal nodes which represent the constant functions 1 and 0. The essential idea behind a BDD comes from the Shannon function expansion which decomposes a Boolean function into two co-factors, \( f(x = 0) \) and \( f(x = 1) \). Hence, a decision node \( v \) in a BDD is associated with a variable \( x_i \) and is connected to two sub-graphs; each represents one of the co-factors, \( f(x_i = 0) \) and \( f(x_i = 1) \). To make the graph compact, any isomorphic sub-graphs are merged. Furthermore, when two child nodes of a decision node \( v \) are the same node, the node \( v \) can be removed. An example circuit and its BDD representation are shown in Figure 2.2.

The size of the BDD is determined both by the represented function and by the chosen

Figure 2.2: An example circuit and its BDD representation. For a decision node \( v \), the dotted line represents \( (v = 0) \) and the solid line represents \( (v = 1) \)
Table 2.1: SAT formula for simple gates

<table>
<thead>
<tr>
<th>Gate function</th>
<th>SAT formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p = \text{AND}(u_1, \cdots u_i)$</td>
<td>$\prod_{j=0}^{i} (u_j + \overline{p}) \cdot \left( \sum_{j=0}^{i} u_j + p \right)$</td>
</tr>
<tr>
<td>$p = \text{NAND}(u_1, \cdots u_i)$</td>
<td>$\prod_{j=0}^{i} u_j + p \cdot \left( \sum_{j=0}^{i} \overline{u_j} + p \right)$</td>
</tr>
<tr>
<td>$p = \text{OR}(u_1, \cdots u_i)$</td>
<td>$\prod_{j=0}^{i} \overline{u_j} + p \cdot \left( \sum_{j=0}^{i} u_j + p \right)$</td>
</tr>
<tr>
<td>$p = \text{NOR}(u_1, \cdots u_i)$</td>
<td>$\prod_{j=0}^{i} \overline{u_j} + p \cdot \left( \sum_{j=0}^{i} u_j + p \right)$</td>
</tr>
<tr>
<td>$p = \text{BUFFER}(u)$</td>
<td>$(\overline{u} + p) \cdot (u + \overline{p})$</td>
</tr>
<tr>
<td>$p = \text{NOT}(u)$</td>
<td>$(u + p) \cdot (\overline{u} + \overline{p})$</td>
</tr>
<tr>
<td>$p = \text{mux}(s, u_1, u_2)$</td>
<td>$(s + u_1 + \overline{p}) \cdot (s + \overline{u_1} + p) \cdot (s + u_2 + \overline{p}) \cdot (s + \overline{u_2} + p)$</td>
</tr>
</tbody>
</table>

ordering of the variables. Depending on the ordering, the number of nodes in a BDD can be linear in the number of variables in the best case, or exponential in the worst case. For some functions (e.g., the multiplication function), the size of the BDDs is always exponential, regardless of variable ordering.

2.2.2 Modelling Circuits in Boolean Satisfiability

Boolean satisfiability is another model that is used as the underlying representation for many CAD applications. This is because advanced techniques, as mentioned in Section 2.1, make SAT solvers powerful and efficient when solving complex CAD problems. Boolean formulae for these CAD problems are derived from the gate-level representation of the design, plus additional constraints for each specific problem. Constructing the Boolean formula for a circuit can be done in linear time [43], and the procedure is as follows. First, each node in the circuit is associated with a Boolean variable. Then, each gate in the circuit is converted into a set of
clauses according to Table 2.1 with the associated Boolean variables. Finally, the Boolean formula of the complete circuit is equal to the conjunction of the Boolean formula of each gate.

**Example 2.4** Revisiting the same circuit shown in Figure 2.2(a), the SAT formula is

\[ \phi = (a + d) \cdot (b + d) \cdot (\overline{a} + b + d) \cdot (\overline{c} + z) \cdot (d + z) \cdot (c + d + z) \]

The first three clauses in \( \phi \) are derived from the AND gate, while the latter three clauses correspond to the OR gate.

To target a specific problem of interest, special clauses are generated to constrain the problem. For example, the clause \( (\overline{z}) \) could be added to \( \phi \) in Example 2.4 to force a logic 0 at the output \( z \) in the example circuit.

### 2.3 Sets of Pairs of Functions to be Distinguished

Given a multi-level network, it is possible to implement the network with different logic functions at some internal nodes such that the global functionality of the network remains the same. In other words, there is an alternative function that can replace the function at the internal node without altering the functionality of the design. The condition of such alternative functions is called *functional flexibility of the node*. Logic synthesis utilizes this flexibility to generate a compact design or to transform nodes in the design for optimization. Sets of Pairs of Functions to be Distinguished (SPFD) is a relatively new representation that provides a powerful formalism to express the functional flexibility of nodes in a multi-level circuit. The concept of SPFD was first proposed by Yamashita et al. [50] for applications in FPGA synthesis.

Intuitively, SPFDs measure how information flows in a Boolean network. Information flows in a network whenever a change in the primary input values causes a node in the network to change its value. Information is said to flow from the primary input to the node. In other words, suppose information required at the output of a node is given; the information must then arrive at one or more fanins of the node. Therefore, an SPFD can be seen as a collection of value transitions that a node function should comply to in order to propagate the information
from the primary input to the primary output. These value transitions express the functional flexibility of a node. Because of the flexibility SPFDs provide, several researchers have applied the concept to logic synthesis and optimization [49, 51–53].

Classically, flexibility of nodes in a Boolean network is expressed with *Incompletely Specified Functions* (ISFs) or *Boolean relations* [4, 54]. An ISF is a function defined over a subset of the complete input space. Inputs that are not defined by an ISF are the combinations of input values of a function that cannot occur. Such combinations are called *don’t cares*. That is, the value of the function can be either logic 0 or logic 1 when these combinations are applied to the inputs. Don’t cares can be categorized into two classes: *Satisfiability Don’t Cares* (SDCs) and *Observability Don’t Cares* (ODCs). SDCs express local input combinations at a node that cannot be generated by the driving logic. ODCs are conditions at the primary inputs such that the output of the node has no effect on the primary outputs of the Boolean network (i.e., no discrepancy can be observed). While calculations of SDCs are straightforward, ODCs are expensive to compute [4, 52]. Hence, in practice, the subset of ODCs, also known as *Compatible Observability Don’t Cares* (CODCs) [55], is normally used. Despite their usefulness, it is shown in [54] that don’t cares are incapable of capturing the complete flexibility of a node, especially for nodes with multiple fanouts.

Boolean relations, on the other hand, can efficiently express the functional flexibility of multiple functions at the same time [54]. A Boolean relation defines a one-to-many mapping
between the inputs and outputs of a multi-output Boolean network, and gives a set of acceptable output patterns for each input minterm. It provides a more general interpretation of flexibility in a network than don’t cares [4]. For example, consider the network depicted in Figure 2.3(a). $x_1$ and $x_2$ are the primary inputs and $o$ is the primary output. The truth table of the local network $N_1$ and the primary output $o$ is shown in Figure 2.3(b). Because of the logic gate XOR, it does not matter whether $\{f_1, f_2\}$ is 00 or 11. Both can result in logic 0 at $o$. Similarly, $\{f_1, f_2\}$ equals 01 or 10 produce the same result at $o$. Such functional flexibility cannot be expressed using ISFs, but can be easily expressed through a Boolean relation, as shown in Figure 2.3(c).

Despite their effectiveness in optimizing a multi-level Boolean network, it is difficult to calculate functional flexibilities and optimize circuits using Boolean relations [56].

2.3.1 Formal Definition of SPFDs

As mentioned earlier, an SPFD can be viewed as a collection of value transitions w.r.t. the primary inputs that the function at a node should comply with. These transitions are represented as pairs of functions, as is formally defined below.

Definition 2.4 Given two logic functions $f$ and $g$, $f$ is said to include $g$, denoted as $g \leq f$, if $f(V)$ always evaluates to 1 for all primary input vectors $V$ where $g(V)$ evaluates to 1.

Example 2.5 Let $g = ab + \overline{a}b$ and $f = a + \overline{b}$. Function $g$ equals 1 when $(a, b) = \{(1, 1), (0, 0)\}$, while function $f$ equals 1 when $(a, b) = \{(1, 1), (1, 0), (0, 0)\}$. Hence, $f$ includes $g$ (i.e., $g \leq f$).

Definition 2.5 A function $f$ is said to distinguish a pair of functions $g_1$ and $g_2$ if either one of the following two conditions is satisfied:

Condition 1: $g_1 \leq f \leq \overline{g_2}$
Condition 2: $g_2 \leq f \leq \overline{g_1}$

Conveniently, one can think of $g_1$ as a subset of the onset of $f$ (denoted $\text{on}(f)$) and $g_2$ as a subset of the offset of $f$ (denoted $\text{off}(f)$) in Condition 1, or vice-versa for Condition 2. That is

Condition 1: $g_1 \leq \text{on}(f) \leq f \leq \overline{\text{off}(f)} \leq \overline{g_2}$
Condition 2: $g_2 \leq \text{on}(f) \leq f \leq \overline{\text{off}(f)} \leq \overline{g_1}$
Chapter 2. Background

Example 2.6 Let \( g_1 = ab \) and \( g_2 = \overline{ab} \). There are four functions that can distinguish the pair \((ab, \overline{ab})\) by satisfying Condition 1, \( g_1 \leq f \leq \overline{g_2} \). The truth table of \( g_1, \overline{g_2} \) and the four functions, \((f_1, f_2, f_3, f_4)\), are shown in Table 2.2(a). According to Condition 1, functions that distinguish the pair \((ab, \overline{ab})\) must generate logic 0 for minterm \( m_1 \) and logic 1 for minterm \( m_3 \), but can generate either logic 0 or logic 1 under minterms \( m_0 \) and \( m_2 \). Such flexibility is the key information provided by SPFDs, as will be discussed later. The complement of these four functions, \((\overline{f_1}, \overline{f_2}, \overline{f_3}, \overline{f_4})\), distinguishes the pair by satisfying Condition 2, as shown in Table 2.2(b).

### Definition 2.6
An SPFD, \( R \), is a set of pairs of functions that need to be distinguished. It can be represented as

\[
R = \{(g_{1a}, g_{1b}), (g_{2a}, g_{2b}), \cdots, (g_{na}, g_{nb})\}
\]

(2.4)

### Definition 2.7
A function \( f \) satisfies an SPFD, \( R = \{(g_{1a}, g_{1b}), (g_{2a}, g_{2b}), \cdots, (g_{na}, g_{nb})\} \), if
Table 2.3: Truth table of functions that can distinguish the pair \((\overline{a}b, \overline{a}b)\)

(a) Condition 1

<table>
<thead>
<tr>
<th>Minterm #</th>
<th>(a)</th>
<th>(b)</th>
<th>(g_1 = \overline{a}b)</th>
<th>(f_1 = \overline{a}b)</th>
<th>(f_2 = a \oplus b)</th>
<th>(f_3 = \overline{a})</th>
<th>(f_4 = \overline{a} + b)</th>
<th>(\overline{f_2} = \overline{a}b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(m_0)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(m_1)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(m_2)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(m_3)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(b) Condition 2

<table>
<thead>
<tr>
<th>Minterm #</th>
<th>(a)</th>
<th>(b)</th>
<th>(g_2 = a \overline{b})</th>
<th>(f_1 = a + b)</th>
<th>(\overline{f_2} = a \oplus b)</th>
<th>(\overline{f_3} = \overline{a})</th>
<th>(\overline{f_4} = a \overline{b})</th>
<th>(\overline{f_4} = a + b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(m_0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(m_1)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(m_2)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(m_3)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\(f\) distinguishes each pair of functions in \(R\):

\[
\{(g_{ia} \leq f \leq g_{ib}) \vee (g_{ib} \leq f \leq g_{ia}) \mid \forall i, \ 1 \leq i \leq n\} \quad (2.5)
\]

**Example 2.7** Assume \(SPFD \ R = \{(ab, \overline{ab}), (\overline{a}b, \overline{a}b)\}\). The truth table of functions that can distinguish the pair \((\overline{a}b, \overline{a}b)\) (based on Condition 1) is summarized in Table 2.3(a). Again, the complement of the four functions, \((f_1, f_2, f_3, f_4)\), in Table 2.3 can also distinguish the pair based on Condition 2, as shown in Table 2.3(b). Functions that satisfy \(R\) have to distinguish both pairs. Hence, according to Table 2.2 and 2.3 there are four functions that can satisfy \(R\): \(a \oplus b\), \(\overline{a} \oplus b\), \(a\), and \(\overline{a}\). It is notable that such functional flexibility cannot be expressed by an ISF, since none of the minterms are don’t cares.

As shown in Example 2.7, there can be multiple functions that satisfy the same SPFD. The idea behind SPFDs is that the internal logic of each node in a design is not fixed. Functions can be changed freely, as long they are within the functional flexibility represented by SPFDs. This
means that manipulating the logic network with SPFDs may require modifying the internal logic accordingly.

Since a minterm is a special function that only evaluates to logic 1 under one specific input combination, an SPFD can be represented as a set of pairs of minterms that have to be distinguished. The definition of SPFDs (Definition 2.6) can be restated in terms of minterms of functions that need to be distinguished.

**Definition 2.8** Given an SPFD, \( R = \{(g_{1a}, g_{1b}), (g_{2a}, g_{2b}), \ldots, (g_{na}, g_{nb})\} \), \( R \) can be represented in terms of minterms of functions as

\[
R = \{(m_i, m_j) | \{m_i \in g_{pa}\} \text{ and } \{m_j \in g_{pb}\}\} \text{ or } \{\{m_i \in g_{pb}\} \text{ and } \{m_j \in g_{pa}\}\}, 1 \leq p \leq n \} 
\]

(2.6)

Therefore, Definition 2.7 can be interpreted as follows: for each minterm pair \((m_a, m_b) \in R\), functions that satisfy \( R \) must be evaluated to different values for \( m_a \) and \( m_b \). What this says in terms of information flow is that the value of functions should change to reflect the change in the values of the input such that information can be propagated through the node.

**Example 2.8** Let \( m_{ij} \), where \( i, j \in \{0, 1\} \), denotes the minterm \( a = i, b = j \) in the input space \( \{a, b\} \). The SPFD \( R \) in Example 2.7 can be expressed in terms of minterms as follows:

\[
\{(m_{11}, m_{01}), (m_{00}, m_{10})\}
\]

Functions that satisfy \( R \) need to return different values for minterms \( m_{11} \) and \( m_{01} \), and for minterms \( m_{00} \) and \( m_{10} \). Taking \( f = a \oplus b \) as a demonstration, \( f(m_{11}) = 0, f(m_{01}) = 1 \) and \( f(m_{00}) = 0, f(m_{10}) = 1 \). Therefore, \( a \oplus b \) can satisfy \( R \).

### 2.3.2 Graphical Representation of SPFDs

In [52], Sinha et al. present SPFDs as graphs. This graphical representation makes it possible to visualize SPFDs and can explain the concept of SPFDs more intuitively. Furthermore, computing SPFDs involves graph manipulation, which is easier to compute.

Based on Definition 2.8 representing an SPFD as a graph is fairly straightforward. The graphical representation of an SPFD contains one vertex for each minterm that needs to be
distinguished by the SPFD. For an $M$-input circuit, SPFDs of nodes in the circuit contain up to $2^M$ vertices. Two vertices are connected by an edge if the function evaluates to complementary values for these two minterms. These edges are referred to as SPFD edges.

Formally, an SPFD $R = \{(m_1a, m_1b), (m_2a, m_2b), \ldots , (m_na, m_nb)\}$ can be represented as a graph $G = (V, E)$, where

$$V = \{m_{ij}, 1 \leq i \leq n, j = \{a, b\}\}$$

$$E = \{(m_{ia}, m_{ib}), 1 \leq i \leq n\} \quad (2.7)$$

**Example 2.9** Figure 2.4 depicts the graph representation of the SPFD $R$ in Example 2.8. Since there are two inputs, $(a, b)$, the graph contains four vertices that represent minterms $\{m_00, m_01, m_10, m_11\}$. Two edges are added for the two pairs of minterms in $R$.

Using the graphical representation, Definition 2.7 can be restated in the following.

**Definition 2.9** A function $f$ satisfies an SPFD $R = (V, E)$ if

$$f(m_i) \neq f(m_j), \text{ for each edge } (m_i, m_j) \in E$$

In graph-theory terms, $f$ has to be a valid coloring of the SPFD graph of $R$; any two nodes connected by an edge must be colored differently. Since only Boolean networks (networks wherein the value of a node is either 1 or 0) are considered in this dissertation, the SPFD is bipartite. That is, only two colors are required to color the SPFD. Note that SPFDs do not specify the value that a function should be evaluated to for each minterm. Hence, if a function $f$ satisfies an SPFD $R$, the complement of the function, $\bar{f}$, also satisfies $R$. 

2.3.3 Computing SPFDs

SPFDs of nodes can be derived in a multitude of ways, depending on their application during logic synthesis. In the original paper [50], required SPFDs at nodes in a network are computed from the primary outputs in reverse topological order. An SPFD of a node represents the input pairs in which the function at the node must evaluate to different values (i.e., differentiate the input pairs); otherwise, the functionality of the design is incorrect. First, the SPFD of a primary output, \( o \), contains all pairs of minterms, \( \{m_i, m_j\} \), where \( m_i \) is a minterm in \( \text{on}(o) \) and \( m_j \) is a minterm in \( \text{off}(o) \). These pairs express value transition at node \( o_i \). Recall that any value transition at a node must flow from one of its fanins. Hence, the next step is to distribute each value transition to one of the fanins of which values are also changed under the minterms in the pair. This procedure is repeated until the primary inputs are reached. Finally, if a node has multiple fanouts, the SPFD of the node is the union of the SPFDs of its fanout nodes. The following example, borrowed from [50], demonstrates the procedure in greater detail.

Example 2.10 Consider the network shown in Figure 2.5(a). For simplicity’s sake, only a portion of the whole network is shown in the figure. The nodes \( a \), \( b \) and \( c \) are internal nodes that are outputs of the internal network \( N_1 \), \( N_2 \) and \( N_3 \), respectively. All internal networks are driven by the primary inputs, \( \{x_1, x_2, x_3\} \). Dotted lines represent wires to other logic networks not shown.

The truth table of all nodes is shown in Figure 2.5(b). Assume that the minterm 101 is a don’t care to the function at \( f \). The functional flexibility of other internal nodes by ISFs are shown in Figure 2.5(c), where \( D \) denotes don’t cares.

SPFD computation begins from the primary output \( f \). Because \( f \) is a primary output, minterms that are not don’t cares and generate complemented logic values at \( f \) should be distinguished. As a result, the SPFD of \( f \) (SPFD\(_f\)) is shown in Figure 2.5(d). For the purpose of clarity, values of a function under each minterm are shown in the SPFD as well, where black nodes represent logic 1 and white nodes represent logic 0. It should be noted that SPFDs do not specify the values of the function. They merely specify that the function needs to evaluate to different values of the two minterms, which are connected by an SPFD edge. Next, to propagate
(a) A Boolean network

(b) The truth table

(c) Functional flexibility by ISF

Figure 2.5: Example of computing SPFDs
SPFD\textsubscript{f} backwards, the information stored in SPFD\textsubscript{f} is distributed to its fanins, d and e. The SPFD of d and e is shown in Figure 2.5(g) and 2.5(h) respectively. Let \( E_{i \rightarrow j} \) denote the SPFD edge connecting minterm i and j. To see how SPFDs are propagated, take the edge \( E_{000 \rightarrow 001} \) in SPFD\textsubscript{e} as an example. Note that a value transition at the output of a node must be caused by a value transition at the input of a node. Since e is the only fanin that changes its logic value under the minterms 000 and 001, this edge is distributed to SPFD\textsubscript{e}.

When the values of multiple fanins are changed under a pair of minterms, the edge can be distributed to any one of the fanins. Taking edge \( E_{000 \rightarrow 001} \) of SPFD\textsubscript{e} as an example, the values of all three fanins to the node e change under the minterms 000 and 001. Hence, this edge can be distributed to any one. In this case, it is distributed to SPFD\textsubscript{a}. Finally, when there are multiple fanouts, such as c, SPFDs from each fanout are merged. One can see that \( E_{000 \rightarrow 010} \) of SPFD\textsubscript{c}, originated from the fanout to node d and \( E_{001 \rightarrow 011} \) of SPFD\textsubscript{c} is originated from the fanout to node e.

The advantage of SPFDs can be seen by comparing the SPFD of c (Figure 2.5(f)) and the ISF of c (the sixth column of Figure 2.5(e)). The ISF of c specifies that there are eight possible functions: the function can have either a logic 0 or a logic 1 under the minterm \{101, 110, 111\}. On the other hand, the SPFD of c only indicates that the function has to evaluate to different values from the minterm 011 to the minterms, \{001, 100\}, as well as from the minterm 000 to the minterms 010. This results in 32 possibilities.

In both [50] and [52], the authors use BDDs as the underlying engine to compute SPFDs in the network. The main difference between these two approaches is that the former uses BDDs to represent the functions of each node in the design directly, while the latter uses BDDs to implement the graphical representation of SPFDs.

However, like other BDD-based techniques, these approaches are limited in terms of memory as the size of the circuit increases. To alleviate the memory issue, Boolean satisfiability is used in [1] to model SPFDs. Instead of computing SPFDs starting from the primary outputs, the authors of [1] calculate the SPFD of a node \( \eta \) by solving a SAT instance built from a miter-like structure, as shown in Figure 2.6. The structure is then converted into a SAT instance to find minterm pairs that can only be distinguished by \( \eta \). Note that a miter is a two-level structure
that is often used to check the functional equivalence between two sets of signals in pairs. It first compares the two sets of signals pairwise using XORs; then, the outputs of XORs are connected with an OR gate. The output of the miter is 0 if all pairs of signals are functionally equivalent.

The construction of the structure in Figure 2.6 is as follows. It consists of two copies of the logic network, $N$ and $N'$. It feeds the primary inputs, $X$, into the first network, $N$, and the primary inputs, $X'$, to the second network, $N'$. A miter $M_1$ is constructed with the primary outputs of both networks. The output of $M_1$ is a logic 1 if a primary input minterm pair $(x, x')$ can make some primary outputs of $N$ and $N'$ different. Let $TFI(\eta)$ denote the transient fanin cone of $\eta$. $Sep(\eta)$, called a separator, is composed of all the primary inputs not in $TFI(\eta)$, plus the nodes in $TFI(\eta)$ that have a fanout outside $TFI(\eta)$. The separator, $Sep(\eta)$, plus $\eta$ forms a cut to the network. This means that every path from the primary inputs to the primary outputs passes through at least one node in the cut. In this structure, the identical separators of $\eta$ and $\eta'$ are selected in both networks. The output of the separators form the second miter $M_2$. The output of $M_2$ is a logic 1 if the separator can distinguish a primary input minterm pair $(x, x')$. Finally, the output of $M_1$ with the complement of the output of $M_2$ are connected.
with an AND gate. Therefore, the final output is a logic 1 if and only if the primary minterm pair can be distinguished by the primary outputs, but not by the separators. This implies that the minterm pair can only be distinguished by the node $\eta$.

Using this approach to obtain the complete SPFD of a node (i.e., all edges that are uniquely distinguished by the node) requires the enumeration of all solutions to the SAT problem. This can be computationally intensive, since the number of edges may increase exponentially according to the number of primary inputs.

### 2.3.4 Properties of SPFDs

Three properties of SPFDs that are important in the later part of this dissertation are described here.

**Property 2.1** Given a node $\eta_k$ whose fanins are $\{\eta_1, \eta_2, \cdots, \eta_n\}$, the SPFD of $\eta_k$ is a subset of the union of the SPFDs of its fanin nodes [57]:

\[
\bigcup_{i=1}^{n} R_{\eta_i} \supseteq R_{\eta_k} \tag{2.8}
\]

Property 2.1 indicates that the ability of a node to distinguish minterm pairs cannot be better than the ability of all of its fanins. Also, as explained earlier, a value transition at the output of a node must be contributed from one of its fanins. This property is the key to performing logic transformations, as discussed later in Chapter 3.

**Example 2.11** Figure 2.7 depicts the SPFDs at the input and output of a two-input OR gate.
It shows that input a can distinguish (vectors in terms of \( a \) \( b \)) 00 and 01 from 10 and 11, and input b can distinguish 00 and 10 from 01 and 11. The output c can distinguish 00 from 01, 10 and 11.

The next property shows that a function can be derived for a node \( g \) in terms of a set of nodes \( N \) if the SPFDs of \( g \) and \( N \) satisfy Property 2.1.

**Property 2.2** Given a node \( \eta_k \) and a set of nodes, \( N = \{ \eta_1, \eta_2, \cdots, \eta_n \} \), as fans to \( \eta_k \), the function of \( \eta_k \) can be expressed in a sum of products form, where the product terms are constructed in terms of nodes in \( N \). The function can be defined formally as follows:

\[
\eta_k = f(\eta_1, \eta_2, \cdots, \eta_n) = \sum_{i=1}^{2^n} (p_i \beta_i) \tag{2.9}
\]

where \( \beta_i \) is a minterm product of \( N \),

\[
\begin{align*}
\beta_0 &= \overline{\eta_1} \overline{\eta_2} \cdots \overline{\eta_n} \\
\beta_1 &= \overline{\eta_1} \eta_2 \cdots \eta_n \\
&\vdots \\
\beta_{2^n} &= \eta_1 \eta_2 \cdots \eta_n
\end{align*}
\]

and \( p_i \) equals 1 if \( \beta_i \) is in the onset of \( \eta_k \); otherwise, \( p_i \) is 0 [51].

Finally, the next property is the basic property that is used by many techniques to modify the internal logic of a node [50–52].

**Property 2.3** Given a node \( \eta_k \) and its immediate fanin \( IF = \{ \eta_1, \eta_2, \cdots, \eta_n \} \), the fanin node \( \eta_i \in IF \) can be replaced with another node \( \eta_t \), if the SPFD of \( \eta_t \) (\( R_{\eta_t} \)) is a superset of the SPFD of \( \eta_i \) (\( R_{\eta_i} \)). That is, the circuit connection, \( \eta_i \rightarrow \eta_k \), can be replaced by the connection, \( \eta_t \rightarrow \eta_k \).

This property can be derived directly from Property 2.1. Since \( R_{\eta_t} \supseteq R_{\eta_i} \), Property 2.1 will still hold after the replacement and the new network at \( \eta_k \) can be constructed with Property 2.2.
2.4 Debugging Methodologies

Each time a design or a silicon prototype fails to comply with a set of specifications, a debugging problem usually follows. As such, debugging manifests itself in virtually every step of the design cycle. When the design does not meet its power requirements, the engineer has to debug the problem and fix it by optimizing certain portions of the design. When a silicon prototype fails a test, silicon debug identifies the root-cause of the error, which needs to be fixed so that the re-spun prototype passes.

Given an erroneous design, the implementation of the design specification, and a set of input test vectors, debugging examines correct and erroneous test vector responses to identify circuit locations that are potential sources of failure. The set of solutions usually contains the actual or the equivalent error site, that is, other locations where a fix can be performed to rectify the design.

It is worth noting that debugging is a hard NP-Complete problem in which the size of the solution space explodes exponentially to the number of errors in the design as [58]

$$solution \ space = (\# \ of \ nets \ in \ the \ circuit)^\# \ errors$$  \hspace{1cm} (2.10)

This complexity promotes the cause for the development of efficient automated debugging tools and methodologies.

Depending on the underlying engine used to drive the algorithm, debugging techniques can be classified as BDD-based, simulation-based or SAT-based [59].

A BDD-based approach performs diagnosis by generating and solving an error equation based on the functionality of both the correct and erroneous circuits. This method is reviewed more thoroughly later in this dissertation. Although effective for single errors, this method presents memory problems, as it uses BDDs [15] to build the error equation. Furthermore, its applicability to multiple errors has been shown to be limited to cases that depend on their structural proximity.

To overcome the excessive memory requirements of BDD-based approaches, debugging with simulation has been extensively investigated. These methods provide a trade-off between time and memory space as the memory requirements remain polynomial in the input size; however,
these methods may require more time to give an answer.

Simulation-based techniques simulate an error-trace and trace backwards from primary outputs to primary inputs while marking suspect lines using different criteria. For each error-trace, they collect the suspect lines and, since the error is present in each one of them, they intersect the results for all runs. As the number of errors increases, their performance degrades. For this reason, their applicability to sequential circuit debugging has been rather limited.

Recently, the introduction of SAT opened new opportunities for cost-effective automated debugging tools. In this concept, the diagnosis problem is formulated into a Boolean satisfiability instance that can be solved by conventional SAT solvers. Solutions returned by the solver correspond to the potential suspects of the error location. Modelling diagnosis of logic designs with Boolean satisfiability is first presented in [58] for combinational designs and then extended into dealing with sequential circuits in [60]. Extensive literature [61] has shown the added advantages of SAT-based debugging when compared to more traditional simulation- and symbolic-based techniques.

SAT-based debugging techniques are the underlying engine used in this dissertation. In Chapter 5 a SAT-based logic debugging tool is used to identify the locations for applying transformations in the logic netlist. In Chapter 5 SAT-based debug techniques form the backbone of the silicon debug methodologies presented. Details on this technique are further discussed in the next section.

2.4.1 SAT-based Debug Methodology

As discussed in Section 2.2.2 the concept of Boolean satisfiability has been applied in many VLSI CAD problems. The main idea behind solving debugging problems with Boolean satisfiability is to construct a SAT instance of the debugging problem [58] where a modern SAT solver selects a set of locations in the design such that it can assign 0 or 1 to those locations to make the behavior of the erroneous design match the expected behavior. This is achieved by inserting a multiplexer $m_i$ at every gate (and primary input), $\eta_i$. The output of this multiplexer, $m_i$, is connected to the fanouts of $\eta_i$ while $\eta_i$ is disconnected from its fanouts. This construction has the following effect: when the select line $s_i$ of a multiplexer is inactive ($s_i = 0$), the original
gate $\eta_i$ is connected to $m_i$; otherwise, when $s_i = 1$, a new unconstrained primary input $w_i$ is connected. This construction is later constrained with the input vector and the expected output response of the particular error-trace.

The multiplexer is referred to as the *correction model*. It functions as an indicator of whether line $\eta_i$ is the source of the error. A potential correction on $\eta_i$ is indicated by the correction value stored in $w_i$ when the select line $s_i$ is assigned to 1. Since all $s$ and $w$ variables are unconstrained, the SAT solver can assign any value \{0, 1\} to them such that the resulting CNF satisfies the constraints applied by the vectors.

Given an erroneous circuit $C$, test sequences $\mathcal{V}$, and expected output responses $\mathcal{Y}$, the formula $\phi$ for debugging is constructed in the following steps.

1. Insert a multiplexer, $m_i = f(\eta_i, w_i, s_i)$ into $C$ at each error suspect $\eta_i$.

2. Convert the design into CNF, according to Table 2.1.

3. Duplicate the CNF for each vector sequence $v \in \mathcal{V}$ and for each cycle of the sequence.

4. Constrain primary inputs with $\mathcal{V}$ and primary outputs with $\mathcal{Y}$.

5. Solve the formula with a SAT solver to obtain the error locations.

An example is depicted in Figure 2.8. Figure 2.8(a) is the sequential circuit being debugged, while Figure 2.8(b) illustrates the same circuit that is unrolled for two cycles and inserted with MUX’s; every gate and primary input is associated with a multiplexer and an unconstrained input $w_i$. Note that multiplexers that are driven by the same gate in each unrolled copy of the circuit share one select line (e.g., $m_1^1$ and $m_2^1$ share $s_1$). This is because these gates represent the same gate in the original sequential circuit. It is not necessary to differentiate them. If one of them is the source of the error, all the remaining should be selected as well. Furthermore, sharing select lines reduces the number of decisions a SAT solver needs to make. This can reduce the runtime of problem solving.

To constrain the SAT problem to a particular error cardinality, $N$ (i.e., there are $N$ locations in the design that can be erroneous simultaneously), additional logic is required to limit the solver activating at most $N$ select lines. This logic can be modelled using a counter or sorter [35].
Figure 2.8: SAT-based diagnosis construction for sequential circuits

For $N = 1$, a single select line is activated. This indicates that the corresponding location is the error location. The value assigned to $w_i$ is the value that can correct the design for the input vector to constrain the problem. For higher values of $N$, the respective number of multiplexer select lines can be set to 1, which indicates an $N$-tuple error suspect.

### 2.4.2 Hierarchical Diagnosis

Ali et al. [62] extend the SAT-based diagnosis to debug designs in a hierarchical manner to improve the performance and resolution of logic debugging. The debug process consists of several iterations. In each iteration, it only considers modules in the same hierarchical level. This reduces the number of suspects that the debug algorithm needs to consider. As a result, diagnosis of the error can be more efficient. The procedure starts from the top-level of the design and goes deeper into the design hierarchy. Suspect candidates for debugging in each iteration are sub-modules of the modules that are determined to be suspects in the previous iteration. The procedure is repeated until the lowest level of the design hierarchy is reached.
Correction models in hierarchy diagnosis are slightly different from the basic construction, as previously described. Considering a module $M$ with output nodes $\{\eta_1, \cdots, \eta_p\}$, multiplexers $\{m_1, \cdots, m_p\}$ are inserted for each output of $M$. Instead of having $p$ select lines, one select line is shared among $m_i$, $1 \leq i \leq p$. This is because, if the design can be corrected by assigning different values at some outputs of $M$, the solver should select the module as the suspect.

The following example illustrates the concept of debugging using hierarchical information.

**Example 2.12** Figure 2.9(a) shows a design and its hierarchical structure. A situation in which hierarchical diagnosis is applied to this design with two iterations is shown in Figure 2.9(b). This design has three modules at the top level. After the first iteration, module $C$ (grey box) is diagnosed to be the module that may contain the error. Therefore, in the second round, only the sub-modules of module $C$, namely, $C_1$, $C_2$, and $C_3$, are considered suspect candidates. At that round, $C_1$ is identified as the potential module containing the error and the suspect candidate list for the third round consists of $C_a$ and $C_b$ only.

With the hierarchical information of the design, diagnosis can start with a coarse-grain global analysis and the search can be directed to local areas after each iteration. Such a procedure reduces the runtime and memory requirement, since there are fewer candidates that need to be analyzed.
Chapter 3

Automated Logic Restructuring with $a$SPFDs

3.1 Introduction

During the chip design cycle, small structural transformations in logic netlists are often required to accommodate different goals. For example, when a design fails functional verification, it is rectified at the locations that are identified as the source of the error by a debugging program [58, 63]. It has been empirically observed that most design errors involve small structural modifications [29]. Logic transformations are also necessary when performing engineering changes (EC) [64] where specification changes at a higher level of abstraction need to be reflected on a logic netlist that has been already synthesized and optimized. Transformations applied in this process have to be minimized so that the engineering effort already invested can be preserved. In addition, logic transformations are important during rewiring-based post-synthesis performance optimization [18, 19] where designs are optimized at particular internal locations to meet specification constraints.

Although these problems can be resolved by another round of full logic synthesis, directly modifying logic netlists is usually preferable for the following reasons:

- Full-blown logic synthesis is a time-consuming and resource-intensive process. With time-to-market pressures, the engineer change process needs to be completed within a short
turn-around-time so that the chip masks can be made.

- Logic transformations for the aforementioned problems often involve a few structural changes within a constrained internal network in the netlist. The global optimization service offered by full logic synthesis is not necessary.

- Existing logic synthesis tools tend to find an ideal representation of the requested function under the given resources and constraints [16,65]. As such, they may significantly modify the structure of the design, meaning that all optimizations that have been done in the logic netlist are wasted.

Hence, logic restructuring has significant merits when compared to re-synthesis. Unlike other processes, today most of these incremental logic changes are implemented manually. The engineer examines the netlist to determine what changes need to be made and how they can affect the remainder of the design. Changes are tracked and verified at the end of the process.

One simple ad-hoc logic restructuring technique modifies the netlist by using permissible transformations from a dictionary model [29]. This technique is mostly adapted for design error correction [63,66] and has been used in design rewiring as well [18]. The dictionary model contains a set of simple modifications, such as single wire additions or removals. These tools first identify locations where the logic transformation is required (i.e., the suspect location of the error). Next, each modification in the model is applied to the location to check whether it can rectify the design as desired. Obviously, the success of these methods directly depends on the ability of the underlying predetermined dictionary to accommodate the necessary netlist changes.

The work presented in this chapter aims to develop a comprehensive methodology to automate the process of logic restructuring in combinational and sequential circuits. It utilizes the SPFDs discussed in Section 2.3 to identify the functional flexibility of nodes in the design and construct logic transformations algorithmically. That is, there is no restriction of static pre-determined structures. SPFDs are the perfect fit for this task, since they assume the structure of the internal node is not determined. Furthermore, in the past, SPFDs have proved to provide additional degrees of flexibility during logic synthesis [56,67].
However, computing SPFDs can be computationally expensive in terms of runtime and memory, as discussed in Section 2.3. To address this problem, a simulation-based approach to approximate SPFDs is presented. The result is called Approximate Sets of Pairs of Functions to be Distinguished (aSPFDs). aSPFDs approximate the information contained in SPFDs using the results of test-vector simulation. They only include minterms that are explored by the given test vectors. This allows them to keep memory and runtime efficient while taking advantage of most of the benefits of the original representation. Although only approximated information of functions is used for logic transformation construction, experimental results indicate that the proposed methodology can successfully perform logic restructuring for an average of 80% of cases of combinational circuits, while restructuring with the dictionary model in [29] can only achieve 16%.

aSPFD-based logic restructuring consists of two stages. The first stage constructs the respective aSPFDs and identifies the function required at specific circuit node(s) so that the design complies with its specification. Next, using aSPFDs as a guideline, the algorithm searches for the necessary nets to construct the required function. Two approaches are presented to perform this search. The first one is an algorithm using Boolean satisfiability to identify the minimum number of new nets required for the desired transformation. Although optimal, the SAT-based approach may require excessive computation at times. The second algorithm, a greedy approach, is later presented to improve runtime performance. One might think that this would sacrifice optimality, but experiments show that, in most cases, this algorithm returns near-optimal results, a favorable trade-off between performance and resolution.

Extensive experiments confirm the theory of the proposed technique and show that aSPFDs provide an effective alternative to dictionary-based transformations. They return modifications where dictionary-based restructuring fails, increasing the impact of tools for debugging, rewiring, EC, etc. Experiments also show the feasibility of using aSPFDs to restructure sequential designs and designs with multiple errors. For combinational circuits, the proposed approach can identify five times more valid transformations than a dictionary-based one. Nevertheless, since the method bases its results on a small sample of the input test vector space, functional verification must follow to confirm the validity of the proposed transformations. Overall, em-
prirical results show that, in both the combinational and sequential circuits, more than 90% of the first transformations returned by our approach pass formal validation.

The remainder of this chapter is structured as follows. Section 3.2 summarizes previous work. The motivation for this work is given in Section 3.3. Section 3.4 defines approximated SPFDs and the procedures to generate aSPFDs. Section 3.5 presents the transformation algorithms utilizing aSPFDs. Applying transformations at multiple locations is discussed in Section 3.6. Experimental results are given in Section 3.7, followed by the summary in Section 3.8.

3.2 Preliminaries

In this section, previous work on logic transformation is reviewed. In particularly, three common techniques, namely dictionary-based correction, redundancy addition and removal, and rewiring are further discussed. Finally, applications of SPFDs on logic optimization are summarized.

Most research done on logic transformations deals with combinational designs. In [68], the authors insert circuitry before and after the original design so that the functionality of the resulting network complies with the required specification. The pre-logic, inserted between the primary inputs and the original network, re-maps the primary inputs to the input lines of the original design. As a result, the vectors that can exercise the error are avoided. The post-logic is fed with the primary inputs and the primary outputs of the original design. This rectifies the design by overwriting the incorrect output response with the correct one. The main disadvantage of this approach is that the additional circuitry may be too large and can dramatically change the performance of the design.

The next two approaches are similar to the one proposed in this dissertation. The idea is, first, to construct the truth table of the new function that should be implemented at the internal node \( \eta \). Then the new function, \( F_\eta \), can be synthesized based on the truth table.

The first approach is proposed in the concept of engineering change by Ling et al. [2]. It modifies the functionality of internal nodes in the original design such that the modified design can comply with a new specification. This approach constructs a structure, referred to as Cir-
Chapter 3. Automated Logic Restructuring with αSPFDs

Figure 3.1: Miter structures to identify the required function at $\eta$ in [2]

circuitConstrain, as shown in Figure 3.1(a). CircuitConstrain consists the original network, $N$, of which the primary inputs and primary outputs are denoted as $X$ and $O$, respectively, the model netlist and a copy of the transitive fanout cone of $\eta$ ($TFO(\eta)$). Node $\eta$ is disconnected from its original fanins and driven by a primary input, $e$. With this structure, all on-set and off-set minterms, in terms of the primary inputs, of $F_\eta$ can be identified. In order to reuse the internal functions to construct $F_\eta$, another structure, as shown in Figure 3.1(b), is used to identify functional dependency between $F_\eta$ and a set of other internal nodes $\{g_1, g_2, \ldots, g_p\}$. CircuitConstrain$_0$ finds all offset minterms of $F_\eta$, while CircuitConstrain$_1$ finds all onset minterms.

The idea of this structure is to check if there exists a minterm pair assigned to $X$ and $X'$, respectively, such that all pairs $(g_i, g'_i)$, where $g_i$ and $g'_i$ are from CircuitConstrain$_0$ and CircuitConstrain$_1$, respectively, for $1 \leq i \leq p$, are the same. If no such a minterm pair can be found, the function $F_\eta$ can be constructed in terms of $\{g_1, g_2, \ldots, g_p\}$. This can also be explained if one views it from an SPFD angle. As discussed in Section 2.3.4, the change of values at a node must be propagated from one of its fanins. If for every pair of minterms from on($\eta$) and off($\eta$), there is a node in $\{g_1, g_2, \ldots, g_p\}$ which changes its value, then $\eta$ is a function of $\{g_1, g_2, \ldots, g_p\}$. All the structures are solved in SAT. Hence, as one can see, the main drawback of this methodology is the size of the SAT instance. Although an approach to reduce the circuit
size is presented, the final structure can be as many as four times the size of the original in the worst case.

The second approach, proposed by Chang et al. [69], performs logic correction in a gate-level netlist. This approach simulates a set of test vectors to generate the signature of each node in the design. A signature is a bit-vector of simulated values for the node. This approach uses a SAT diagnosis tool to identify the signature that should be generated at \( \eta \) for the given test vectors, referred to as target signature. It states that \( F_\eta \) can be re-synthesized in terms of a set of internal signals \( \{g_1, g_2, \cdots, g_p\} \), if, for every pair of \((0, 1)\) in the target signature, there exists at least one node \( g_i \), where \( 1 \leq i \leq p \), such that the corresponding bits of the signature of \( g_i \) are different as well. Again, this concept is similar to the concept of SPFDs: a value transition at a node requires a value transition at one of its fanins.

SAT sweeping [70,71] is a technique to identify the internal nodes of a design with equivalent functionality. Internal nodes that are functionally equivalent can be merged and can result in a smaller network. Hence, SAT sweeping has been used in functional verification to reduce the problem size and logic synthesis for optimization. However, in the concept of logic restructuring, the application of SAT sweeping is limited. This is because node replacement can only occur when there is an existing node with the same functionality. SAT sweeping is not able to find a solution for cases where multiple internal nodes are required to construct the logic transformation.

The next subsection discusses a dictionary model, which is often used for rewiring, error rectification, etc. Then, two rewiring techniques are reviewed, namely Redundancy Addition and Removal (RAR), and diagnosis-based rewiring. The former constructs transformations similar to those defined in the dictionary model, while the later applies the dictionary model directly for restructuring.

### 3.2.1 Dictionary-Based Correction

A commonly used ad-hoc solution to logic transformations in the concept of design error correction is to use the predetermined error dictionary model of Abadir et al. [29]. This model consists of simple design errors that involve single wire (or gate) additions or removals, as shown
Chapter 3. Automated Logic Restructuring with αSPFDs

Table 3.1: Common design error types [29]

<table>
<thead>
<tr>
<th>Error types</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gate replacement</td>
<td><img src="image" alt="Gate replacement" /></td>
</tr>
<tr>
<td>Missing/extra inverter</td>
<td><img src="image" alt="Missing/extra inverter" /></td>
</tr>
<tr>
<td>Missing/extra input wire</td>
<td><img src="image" alt="Missing/extra input wire" /></td>
</tr>
<tr>
<td>Missing/extra gate</td>
<td><img src="image" alt="Missing/extra gate" /></td>
</tr>
<tr>
<td>Incorrect input wire</td>
<td><img src="image" alt="Incorrect input wire" /></td>
</tr>
</tbody>
</table>

After the diagnosis tool identifies the potential location of an error, each error model is assumed to be the error occurred and corrections that can counter the error are sought. For example, assume that an internal OR gate, \( c = \text{OR}(a, b) \), is diagnosed as the error location. To correct it, different gate types are first tested: \( c = \text{AND}(a, b) \), \( c = \text{XOR}(a, b) \), etc. If none of these gate types rectifies the design, the next error model is considered. Assume the next model is missing input wire; in this case, \( c \) is rectified by adding a new fanin. Hence, all nodes that are not in the transitive fanout cone of \( c \) are tried.

Although it is an ad-hoc solution, this approach is favorable because the transformations included in the model are simple and easy to implement. This is the major advantage of this approach over other formal techniques. However, the success of transformations is limited by the underlying dictionary model. As will be shown later, this model may not be adequate if more complex modifications are required.

3.2.2 Redundancy Addition and Removal

Rewiring is a post-synthesis logic optimization technique that replaces one wire with another in order to improve the performance of a design (things such as area or power consumption), without altering the functionality of the design. Redundancy addition and removal achieves this through the iterative addition and removal of redundant wires [19]. A wire in a netlist is
Chapter 3. Automated Logic Restructuring with $\alpha$SPFDs

Figure 3.2: Example of a circuit with redundancy connections

(redundant if the functionality of the netlist remains the same after the wire is replaced with a logic 0 or a logic 1. For example, consider the simple circuit shown in Figure 3.2(a), one can verify that the wire $c \rightarrow e$ can be replaced with a logic 1 (Figure 3.2(b)) and the function of $o$ will remain the same. Hence, the wire $c \rightarrow e$ is redundant. Finally, since the gate $e$ is an AND gate, the constant 1 can be removed. Figure 3.2(c) depicts the simplified circuit.

Test generation for stuck-at faults is often used for redundancy identification. A stuck-at fault is a common fault model used in fault simulation or fault diagnosis to represent physical defects. For instance, short-to-ground (short-to-power) is modelled with stuck-at-0 (stuck-at-1). A node $\eta$ is stuck-at-$v$, meaning the logic value of $\eta$ is stuck at logic $v$. A stuck-at-$v$ on $\eta$, denoted as $sa-v(\eta)$, can be detected by an input vector if (a) $\eta$ has a logic value $\overline{v}$ under the vector, and (b) a logic change at $\eta$ can be observed at some primary outputs. If no test exists for a given stuck-at fault, $sa-v(\eta)$, it means that the functionality of the circuit does not change in the presence of the fault. Hence, the fault is redundant and $\eta$ can be replaced with a logic $v$.

RAR technique adds a redundant wire such that one or many existing irredundant wires become redundant under the presence of the new added wire. As a result, these new redundant wires can be safely removed without altering the functionality of the design. Example 3.1, taken from [19], illustrates the RAR procedures.

Example 3.1 Consider the circuit in Figure 3.3(a) where wire $w_r = k \rightarrow p$, indicated by a dotted line, is not part of the original netlist. Wire $w_t = g \rightarrow j$, indicated by a bold line, is the target wire to be removed. It can be removed if the fault $sa-1(w_t)$ is not detectable (then it can be replaced with a logic 1). RAR first identifies the necessary assignments to detect $sa-1(w_t)$. Input vectors to detect $sa-1(w_t)$ need to set $g$ to 0. Also, it is necessary to set $c = 1$, $f = 1$, $r = 1$, $k = 1$.

(a) Original circuit with redundancy connections (b) Replace the redundant connection with a constant 1 (c) Simplified circuit
and $m = 0$ to propagate the fault to the primary output $o_2$. These assignments imply $h = 0$, and $k = 0$. Since gate $p$ is on the only fault propagation path, the fault becomes undetectable if a wire from $k$ to $p$ is made (as the output of $p$ is forced to 0 by $k = 0$). Hence, $w_t$ becomes redundant once $w_r = k \rightarrow p$ is added. To ensure that adding $w_r$ does not alter the functionality of the circuit, RAR verifies the redundancy of $w_r$ by checking whether the fault $sa-1(w_r)$ is detectable. One can verify that this fault is undetectable and $w_r$ is redundant. As a result, $w_t$ can be removed. Furthermore, the wire $l \rightarrow m$ also becomes redundant in the presence of $w_r$. The final circuit after adding and removing these redundancies is shown in Figure 3.3(b).

Other than using stuck-at fault test generation, several heuristics are proposed to help identify redundant wires [11, 12, 72]. However, not all irredundant wires can be successfully removed because redundant wires may not be found. The authors in [73] extend the traditional RAR technique by removing the irredundant wire first, followed by the addition of a new
irredundant wire to rectify the functionality of the circuit. Nevertheless, all logic restructuring operations performed by RAR techniques are limited to single wire additions and removals, similar to the transformations in Table 3.1. There is little success in trying to add and remove multiple wires simultaneously due to a large search space and complicated computation [74].

3.2.3 Diagnosis-Based Rewiring

RAR discussed in the previous session was a popular approach in the 1990s until the work of [18] was introduced. In [18], the authors view the rewiring problem from a logic debugging angle. Unlike RAR, this method has been shown to exploit the complete solution space and, hence, it offers more flexibility in optimizing a design and achieving larger performance gains. It has been also proven that computationally, the method is, in theory, as expensive as the traditional RAR techniques but in practice it may run even faster.

The general flow of the diagnosis-based rewiring algorithm proceeds as follows. First, the algorithm identifies the target wire $w_t$ and introduces a design error by eliminating $w_t$. Next, the design rewiring method uses a debugging algorithm to return some equivalent corrections. For each equivalent correction, the optimization gain is computed and the logic transformation with the highest improvement is kept.

Example 3.2 Consider the same circuit in Example 3.1. Under the application of diagnosis-based rewiring, the target wire $w_t$ in Figure 3.3(a) is removed from the design. This introduces a design error (i.e., missing input) to the circuit. Next, the design error diagnosis and correction algorithm in [75] is used to rectify the modified circuit. It identifies the potential locations of design errors and rectifies each location with the dictionary model discussed in Section 3.2.1. In this example, the actual error (i.e., the missing wire from $g$ to $j$) and the missing wire from $k$ to $p$ are identified. The latter logic transformation is the same transformation performed by RAR.
3.2.4 Logic Rewiring for Sequential Circuits

Unlike combinational restructuring, there is less previous work dealing with logic transformation in sequential circuits. The main reason for this is that sequential circuits can be hard to restructure due to the presence of memory elements. For this reason, the Boolean function of a net may depend on many previous states of the design. This increases the complexity of the underlying analysis severely.

There are two main approaches to sequential logic optimization: Retiming and Resynthesis (RR) [10] and Sequential Redundancy Addition and Removal (SRAR) [19]. The former technique consists of moving the flip-flops across combinational gates, merging combinational blocks and optimizing the resulting logic with combinational techniques. The latter technique extends the combinational Redundancy Addition and Removal to sequential designs by considering sequential redundancies. The authors in [76] show that the sequential transformations that can be obtained by RR can also be obtained with SRAR.

3.2.5 SPFD-Based Optimization

Extensive research work has applied SPFDs to perform logic optimization. Sinha et al. in [52] utilize SPFDs to determine if any of the fanin wires of a node can be removed. The algorithm computes the minimum information the SPFD of each fanin should contain. That is, the minterm pairs that need to be distinguished by the output of the node, but can not be distinguished by any other fanins. Such a SPFD is referred to as minimum SPFD. If the minimum SPFDs of some fanin wires are empty, those wires can be removed. For wires with non-empty minimum SPFDs, an algorithm is proposed to find alternative wires such that the new function is smaller in terms of the number of literals. The wire replacement only occurs locally; that is, the target wire and the alternative wire drive the same node. Cong et al. in [77] extend the work to find alternative wires that might be far away from the target wire. However, the transformations are still limited with a single wire addition and removal.

Finally, a sequential state encoding optimization using SPFDs is proposed in [49]. The authors first define sequential SPFD: each vertex of a sequential SPFD represents one state of the sequential design; an edge exists between two vertices if the output response of the two
states represented by the vertices should be different. To avoid the input vector space explosion issue, as mentioned in Section 2.2, the algorithm progressively partitions the state space into equivalence classes until no additional refinements can be made. Consequently, states in an equivalence class can be merged and a new state encoding can be applied. A procedure is presented to re-synthesize the original circuit with the new states. The authors conclude that the size of the input space remains a major challenge for some circuits.

3.3 Motivation

The work presented in this chapter is motivated by the following empirical observation. As mentioned in Section 3.2, most common approaches for logic transformation use a dictionary model similar to the one shown in Table 3.1. However, such a predetermined dictionary model, although effective in several cases, may not be adequate when complex transformations are required (for instance, the addition or deletion of multiple gates and wires).

A set of experiments is conducted to evaluate the performance of the general purpose dictionary model in [29]. The experimental result is compared against the result of a formal technique called error equation.

3.3.1 Error Equation

Error equation [63] is a symbolic methodology that answers with certainty whether there exists a modification that transforms an old design $C_e$ to have the same functionality as the reference design $C_c$ at a specified circuit location $\eta_k$. The concept of the error equation approach can be clearly explained as a miter structure, as shown in Figure 3.4. The miter compares the primary outputs of the reference design and the original netlist, where $\eta_k$ is driven by a new primary input. The output of the miter is equal to 0 when both netlists have the same output responses under the same input minterms. That is, there exists a function at $r_k$ that can make the original netlist behave in the same was as the reference design. The function of the output of the miter is denoted as $EQ(X, r_k)$ and can be formally defined as
Chapter 3. Automated Logic Restructuring with αSPFDs

Figure 3.4: Circuitry of the error equation

\[
EQ(X, r_k) = \sum_{o_i \in O} (f^c_{o_i}(X) \oplus f^e_{o_i}(X, r_k)) = 0
\] (3.1)

where \( f^c_{o_i}(X) \) is the function of the primary output \( o_i \) of the reference design \( C_c \) in terms of the primary inputs, and similarly \( f^e_{o_i}(X, r_k) \) is the function of the primary output \( o_i \) of the original design \( C_e \) in terms of the primary inputs and \( r_k \). The error equation is \( EQ(X, r_k) = 0 \).

Intuitively, a solution to the error equation is a function for \( r_k \) that depends on \( X \) and when implemented on \( \eta_k \), the function makes the respective primary outputs of \( C_c \) and \( C_e \) implement the exact same functions. If such a solution does not exist, it simply means that the circuit cannot be rectified with any modifications at \( \eta_k \).

Solutions to the error equation, \( f'_{\eta_k}(X) \), are incompletely specified functions in terms of the primary inputs:

\[
EQ(X, r_k = 0) \leq f'_{\eta_k}(X) \leq \overline{EQ(X, r_k = 1)}
\] (3.2)

where \( f(x) \leq g(x) \) for Boolean functions \( f, g \) if and only if \( f(x)g(x) = 0 \). Any function in the above range is guaranteed to make \( C_e \) functionally equivalent to \( C_c \) when implemented at \( \eta_k \).

Note that values of \( X \) that evaluate \( EQ(X, r_k = 0) \) equals to 0 is the offset of the solution, while values of \( X \) that evaluate \( EQ(X, r_k = 1) \) equals to 0 is the onset of the solution. This equation is equivalent to the idea in SPFDs where the solution to the error equation has to distinguish the function pair \( (EQ(X, r_k = 0), EQ(X, r_k = 1)) \).
In the context of this dissertation, the notion of an error equation is used to identify whether a particular error location has a correction or not. This can be done by validating the equation below:

$$EQ(\mathcal{X}, r_k = 0) \cdot EQ(\mathcal{X}, r_k = 1) = 0$$

for all assignments to $\mathcal{X}$  \hspace{1cm} (3.3)

Equation (3.3) fails only if there exists a minterm such that no matter what value is assigned to $r_k$, the output response of the original circuit cannot match with the output response of the reference circuit. In other words, no function at $\eta_k$ can make the two designs functionally equivalent. This condition can be easily checked using BDDs: constructing Equation (3.3) with BDDs and checking if all paths from the root reach the terminal node 0 in the BDD.

### 3.3.2 Limitations of the Dictionary Model

The following experiment studies the effectiveness of dictionary-based transformations. The experiment is set up as a design error diagnosis and correction problem. Random design errors are inserted into ISCAS’85 benchmarks. The tool in [75] is used to diagnose the erroneous circuit. It implements a simple path-trace simulation-based diagnosis method to identify single locations that are potential candidates for rectification. Then, it applies the dictionary model in [29] to construct corrections for each location.

In this study, the effectiveness of the dictionary model is measured against the effectiveness of the error equation approach described in the previous section. If there is no solution to the error equation constructed for a target node, no transformation on that location can correct the design. This can happen because the diagnosis tool identifies potential error locations based on a subset of input vectors. In this experiment, 1000 vectors are used. Therefore, these locations are guaranteed to be correctable for the given set of vectors only. Since the error equation approach is a formal approach that checks the complete input space, some of these locations returned by the diagnosis tool may not in fact be correctable.

Results are summarized in Table 3.2. Each number is the average of 10 single error experiments per circuit.

The first column shows the name of the benchmark. Two types of error complexities are
Table 3.2: Quantifying logic transformations

<table>
<thead>
<tr>
<th>Circ.</th>
<th>Error location</th>
<th>Error equat.</th>
<th>Dict. model</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_{432}$s</td>
<td>9.8</td>
<td>75%</td>
<td>44%</td>
</tr>
<tr>
<td>$c_{499}$s</td>
<td>7.1</td>
<td>76%</td>
<td>40%</td>
</tr>
<tr>
<td>$c_{880}$s</td>
<td>3.8</td>
<td>67%</td>
<td>38%</td>
</tr>
<tr>
<td>$c_{1355}$s</td>
<td>5.3</td>
<td>100%</td>
<td>19%</td>
</tr>
<tr>
<td>$c_{1908}$s</td>
<td>18.0</td>
<td>84%</td>
<td>23%</td>
</tr>
<tr>
<td>$c_{2670}$s</td>
<td>9.2</td>
<td>99%</td>
<td>11%</td>
</tr>
<tr>
<td>$c_{5135}$s</td>
<td>6.4</td>
<td>100%</td>
<td>25%</td>
</tr>
<tr>
<td>$c_{3540}$c</td>
<td>3.0</td>
<td>100%</td>
<td>6%</td>
</tr>
<tr>
<td>$c_{5315}$c</td>
<td>6.4</td>
<td>97%</td>
<td>16%</td>
</tr>
<tr>
<td>$c_{7552}$c</td>
<td>20.6</td>
<td>64%</td>
<td>20%</td>
</tr>
</tbody>
</table>

inserted into each benchmark. A “simple” error (labelled with suffix “s”) involves changes on a single gate. Changes can be (a) a change of gate type, (b) the addition or removal of a fanin, or (c) the replacement of a fanin with another node. A “complex” error (labelled “c”) applies multiple changes to a single gate and nodes in the fanin cone of that gate. Both locations and changes are randomly selected.

The second column of Table 3.2 contains the average number of error locations returned by path-trace. The following two columns show the percentage of error locations that can be fixed with error equation and with the exhaustive dictionary-based rectification method [75]. Comparing numbers in the third and fourth columns, it can be seen that, in six out of ten circuits, locations where the dictionary-model approach can rectify is 30% less than locations where the error equation approach can. Taking $c_{1908}$s as an example, the error equation returns that modification on 15 locations (84% of 18.0 locations) can rectify the design, whereas the dictionary model approach is successful in only four locations. Furthermore, the success of the dictionary model approach diminishes further when more complex re-synthesis is required. This is because complex modifications perturb the functionality of the design in ways that simple dictionary-driven transformations may not be able to address. Such modifications are common
in today’s intricate design environment where errors or changes in RTL necessitate complex local changes in the netlist [64]. Automated logic transformation tools that can address these problems effectively are desirable to increase the impact of the underlying engines for debugging, rewiring, EC, etc.

### 3.4 Approximating SPFDs

As discussed in Section 2.3, SPFDs are traditionally implemented with BDDs or with SAT. Each representation has its own set of advantages and disadvantages. Computing SPFDs involves Boolean function manipulation. Using BDDs to represent the functionality of nodes in the design can make those operations more efficient. However, BDD representations of large designs and of some types of circuits (e.g., multipliers) may not be memory efficient. The size of BDDs for those circuits can be exponential in the number of the primary inputs [78]. As discussed in Section 2.3.3, SAT-based SPFD computation constructs a SAT instance of a simple miter-like structure to identify the minterm pair that needs to be distinguished by a node in the design. Although this approach alleviates the memory issue with BDDs, it can be computationally intensive to obtain all the minterm pairs that need to be distinguished.

Intuitively, the runtime and memory overhead of formal approaches can be reduced if fewer SPFD edges (or minterms) are captured by the formulation. Hence, this section presents a simulation-based approach to “approximate” SPFDs to reduce the information that needs to be processed. The main idea behind αSPFDs is that they only consider a subset of minterms that are important to the problem. Although αSPFDs are based on a small set of the complete input test vector space, experiments show that αSPFDs provide enough information to construct valid transformations in most cases.

Clearly, the selection of minterms is crucial to the quality of αSPFDs. Recall the diagnosis-based rewiring technique discussed in Section 3.2.3 has shown that logic restructuring can be effectively viewed as a pair of “error/correction” operations. As such, the required transformation simply corrects an erroneous netlist to a new specification. From this point of view, test vectors can be thought of as a way to describe the erroneous behavior of the design. Hence, it
is constructive to see that test vectors that diagnosis uses to locate the error (and its equivalent faults) are a good means of selecting minterms that should be included in αSPFDs for logic restructuring. That is, minterms explored by test vectors are more critical than others. The following subsections demonstrate the steps to constructing αSPFDs using a set of primary input vectors.

From the viewpoint of the “error/correction” operation pair for logic restructuring, two circuits $C_e$ and $C_c$ with the same number of the primary inputs and the primary outputs are considered. Let $V = \{v_1, \ldots, v_q\}$ be a set of input vectors. For combinational circuits, each $v_i \in V$ is a single vector, while for sequential circuits, each $v_i$ is a sequence of input vectors. Test vector set $V$ can be divided into two subsets, $V_e$ and $V_c$, such that $C_c$ and $C_e$ have different (same) output responses when $V_e$ ($V_c$) is simulated. Let $\eta_{err}$ be the node in $C_e$ where the correction is required, such that $C_e$ is functionally equivalent to $C_c$ after the rectification. Node $\eta_{err}$ can be identified using diagnosis [58, 63] or formal synthesis [64] techniques, and is referred to as a transformation node in the remaining discussion.

Next, an αSPFD is defined as follows:

**Definition 3.1** Let $M$ consist of all primary input minterms and $M'$ be a subset, where $M' \subseteq M$, the approximate SPFD (αSPFD) $R_{\eta_i}^{appx}$ w.r.t $M'$ of a node $\eta_i$ specifies the minterm pairs in $M' \times M'$ that $\eta_i$ has to distinguish. $R_{\eta_i}^{appx}$ contains no information about the minterms in $M - M'$.

In other words, the αSPFD of a node considers what needs to be distinguished only for a subset of the primary input vectors. Since it stores less information than the SPFD of the node, it is inherently less expensive to represent, manipulate and compute.

The next two subsections present the procedures used to compute αSPFDs using a test set $V$ and a SAT solver for nodes in combinational and sequential circuits, respectively. In this section, it is assumed that there is only one transformation node. The case of simultaneously restructuring multiple locations is discussed in the later section. Note that information included in the αSPFD of transformation node $\eta_{err}$ is different than information included in the αSPFDs of the remaining nodes that are not in $TFO(\eta_{err})$. The αSPFD of $\eta_{err}$ captures information
changes that the new function at $\eta_{err}$ should include such that the design with the new function at $\eta_{err}$ can comply with the reference design. On the other hand, $a$-SPFDs of the remaining nodes that are not in $TFO(\eta_{err})$ represent the information changes occurred at those nodes, that is, the ability of the function of each node to distinguish minterm pairs. Therefore, approaches to computing $a$-SPFDs for these two types of nodes are different. To avoid combinational feedback, nodes in $TFO(\eta_{err})$ cannot be used for restructuring; hence, their $a$-SPFDs do not need to be computed.

### 3.4.1 Computing $a$-SPFDs for Combinational Circuits

As discussed earlier, the $a$-SPFD of the transformation node $\eta_{err}$ should contain the pairs of primary input minterms that need to be distinguished by the new function of $\eta_{err}$. To identify those pairs, the correct values of $\eta_{err}$ under the test vectors $V$ are first identified. Those values are what the function represented by $\eta_{err}$ should evaluate for $V$ after the restructuring is implemented. Such a set of values is referred to as expected trace, denoted as $E_T$.

**Definition 3.2** Given an erroneous circuit $C_e$, test vectors $V$, and their corresponding expected output response $Y$, the expected trace ($E_T$) of node $\eta_{err}$ is a set of values at $\eta_{err}$ for each vector in $V$ such that $C_e$ produces outputs identical to $Y$ when the value is forced at $\eta_{err}$.

After the expected trace of $\eta_{err}$ is calculated, the procedure uses the trace to construct $a$-SPFDs of $\eta_{err}$ and for the remaining nodes. In practice, $V$ includes vectors that detect errors, as well as ones that do not. This is because both types of vectors can provide useful information about the required transformation. While erroneous vectors provide information regarding changes the new function should achieve, the correct vectors provide information that the new function should retain. Both are helpful in defining the requirements of logic transformations.

Details of the procedure to generate $a$-SPFDs of nodes in combinational circuits $C_e$ w.r.t. test vector $V = \{V_c \cup V_e\}$ are as follows:

**Step 1.** Simulate $C_e$ with the input vector $V'$.

**Step 2.** Let $V_c(\eta_{err})$ and $V_e(\eta_{err})$ denote the value of $\eta_{err}$ when $C_e$ is simulated with $V_c$ and $V_e$, respectively. Since $\eta_{err}$ is the only location where transformation will be applied, the
new function at \( \eta_{\text{err}} \) has to evaluate to the complemented value of \( V_e(\eta_{\text{err}}) \) in order to eliminate the discrepancy at the primary outputs. Hence, the expected trace of \( \eta_{\text{err}} \), denoted by \( E_{T}^{\eta_{\text{err}}} \), is \( \{ V_e(\eta_{\text{err}}), V_c(\eta_{\text{err}}) \} \) for vectors \( \{ V_e, V_c \} \).

Step 3. The \( a \)SPFD of \( \eta_{\text{err}} \) states that minterms in \( \text{on}(E_{T}^{\eta_{\text{err}}}) \) have to be distinguished from minterms in \( \text{off}(E_{T}^{\eta_{\text{err}}}) \). Hence, \( R^{\text{appx}}_{\eta_{\text{err}}} \) contains an edge for each pair \( (a, b) \in \{ \text{on}(E_{T}^{\eta_{\text{err}}}) \times \text{off}(E_{T}^{\eta_{\text{err}}}) \} \).

Step 4. For each node \( \eta_k \notin \{ TFO(\eta_{\text{err}}) \cup \eta_{\text{err}} \} \), the \( a \)SPFD of \( \eta_k \) distinguishes each pair \( (a, b) \in \{ \text{on}(\eta_k) \times \text{off}(\eta_k) \} \) under the test vectors \( \mathcal{V} \). This is because the SPFDs of those nodes are not changed after restructuring. Since re-synthesis cannot use nodes in \( TFO(\eta_{\text{err}}) \) as they create combinational feedback, it is not necessary to compute the \( a \)SPFDs for these nodes.

The following example demonstrates the procedure described above.
Example 3.3 Figure 3.5(a) depicts a circuit; its truth table is shown in Figure 3.5(b). Let the wire \( e \rightarrow z \) (the dotted line) be the target to be removed. After the removal of \( e \rightarrow z \), an erroneous circuit is created where the new \( z \), labelled \( z_{\text{mod}} \), becomes \( \text{AND}(\overline{d}, f) \). The value of \( z_{\text{mod}} \) is shown in the eighth column of the truth table (Figure 3.5(b)).

Suppose the design is simulated with test vectors \( \mathcal{V} = \{001, 100, 101, 110, 111\} \). The discrepancy is observed when the vector 110 is applied. Therefore, \( \mathcal{V}_e = \{110\} \) and \( \mathcal{V}_c = \{001, 100, 101, 111\} \). Let \( z_{\text{mod}} \) be the transformation node. As described in Step 2 of the procedure, \( V_e(z_{\text{mod}}) = \{1\} \) for \( \mathcal{V}_e \), and \( V_c(z_{\text{mod}}) = \{0, 1, 0, 0\} \) for \( \mathcal{V}_c \). Hence, the expected trace of \( z_{\text{mod}} \) consists of the complement of \( V_e(z_{\text{mod}}) \) and \( V_c(z_{\text{mod}}) \) as shown in the final column of Figure 3.5(b). Finally, the aSPFD of \( z_{\text{mod}} \) w.r.t. \( \mathcal{V} \) is generated according to the expected trace. It contains four edges, between \( \{100\} \) and \( \{001, 101, 110, 111\} \), as shown in Figure 3.6(a). The black (white) vertices indicate that \( z_{\text{mod}} \) has a logic value 1 (0) under the labelled minterm. The dotted vertices indicate that the labelled minterm is a don’t care w.r.t. \( \mathcal{V} \). This aSPFD is approximated since it only contains a subset of the complete information about minterms that \( z_{\text{mod}} \) needs to distinguish and maintain correct functionality. For comparison, the SPFD of \( z_{\text{mod}} \) is shown in Figure 3.6(b). One can see that information included in aSPFD of \( z_{\text{mod}} \) is much less than what the SPFD representation includes. The minterms that are not encountered during the simulation are considered don’t cares in aSPFDs. For instance, because the vector 000 is not simulated, the aSPFD of \( z_{\text{mod}} \) does not distinguish 110 from 000. Due to the loss of information, transformations identified by the procedure may fail to correct designs. However, experimental results show that high percentages of identified logic transformations are valid. This observation is justified in the later section.

Computing aSPFDs of nodes other than the transformation node is fairly straightforward. Taking node \( b \) as an example, the aSPFD of \( b \) for the same set of input vectors consists of edges between every pair of minterms in \( \text{on}(b) = \{110, 111\} \) and \( \text{off}(b) = \{001, 100, 101\} \) w.r.t. \( \mathcal{V} \). The aSPFD of \( b \) is shown in Figure 3.6(c).
3.4.2 Computing aSPFDs for Sequential Circuits

The procedure of building aSPFDs of nodes in sequential circuits is more complicated than the procedure for combinational circuits due to the state elements. Each node in the circuit not only depends on the present value of the primary inputs, but also on values applied to them in previous timeframes. This time-dependency characteristic prohibits the application of the procedure of generating aSPFDs presented in Section 3.4.1 directly to sequential circuits.

First of all, the expected trace of the transformation node $\eta_{err}$ is not simply the complemented values of the node under the erroneous vector sequences. The complemented values are not guaranteed to fix the error, because it is not known in which timeframe the error condition is excited. Complementing values in all timeframes risks the introduction of more errors.

Moreover, recall that sequential circuits are modelled in the ILA representation. The value of nets in the circuit at $T_i$ for some input vector sequences is a function of the initial state input and the sequence of the primary input vectors up to and including cycle $T_i$ (i.e., $f(S^1, X^1, \cdots, X^i)$). This implies that the input space that needs to be distinguished by the aSPFD at $\eta_{err}$ is different across timeframes. Each of these aSPFDs is equally important and a valid transformation at $\eta_{err}$ has to satisfy all of them. To simplify the complexity of the problem, one aSPFD that integrates information stored in the aSPFD in each timeframe is constructed.

The proposed procedure generates such a unified aSPFD over the input space $\{S \cup X\}$. It first determines the values of the state elements in each timeframe for the given set of input vectors that should take place after the transformation. Then, a partially specified truth table in terms of the primary input and the current states of the new function at $\eta_{err}$ can be generated. The aSPFD of $\eta_{err}$ over the input space $\{S \cup X\}$ is constructed based on the truth table. The complete procedure is summarized below:

Step 1. Extract the expected trace $E_T$ of $\eta_{err}$ for an input vector sequence $v$. This can be done by constructing a satisfiability instance with the unrolled circuit $C_e$ as follows. Given the expected output response $Y$ for $\mathcal{O}$ under $v$, the formula is expressed as

$$\Phi = \prod_{i=0}^{k} \Phi_{C_e}^{i}(v^i, Y^i, \eta_{err}^i)$$  \hspace{1cm} (3.4)
Each \( \Phi_i^e \) represents a copy of \( C_e \) at timeframe \( i \), where \( \eta_i^\text{err} \) is disconnected from its fanins and treated as a primary input. The original primary inputs of \( C_e^i \) are constrained with \( v^i \) and the primary outputs of \( C_i \) are constrained with \( y^i \). Since \( N = \{\eta_0^\text{err}, \ldots, \eta_k^\text{err}\} \) are the only free variables in \( \Phi \), the SAT solver needs to assign values to \( N \) to make \( C_e \) comply with the expected responses. Hence, if the SAT solver can find a valid assignment satisfying \( \Phi \), the values assigned to \( N \) are the desired values of \( \eta_i^\text{err} \) for \( v \).

**Step 2.** Simulate \( C_e \) with \( v \) applied at primary inputs and \( E_T \) applied at \( \eta_i^\text{err} \) to determine state values in each timeframe. Since the expected trace contains the values that the new transformation should evaluate to, those state values obtained from simulation are also what should be expected after the transformation is applied. As a result, minterms in terms of \( \{X \cup S\} \) can be extracted; a partially specified truth table of the transformation that should be implemented at \( \eta_i^\text{err} \) in \( C_e \) can be constructed.

**Step 3.** The aSPFD of a node \( \eta \) in \( C_e \) contains an edge for each minterm pair in \( \{\text{on}(\eta) \times \text{off}(\eta)\} \) according to the partially specified truth table, where minterms are in terms of \( \{X \cup S\} \).

**Example 3.4** Figure 3.7(a) depicts a sequential circuit unrolled for three cycles under the simulation of a single input vector sequence. Assume the correct response at \( o^1 \) should be 0 and net \( p \) is the transformation node. To determine the expected trace of \( p \), \( ps \) are made into primary inputs, as shown in Figure 3.7(b), and construct a SAT instance from the modified circuit, the input vector \( \{10, 01, 10\} \), and the expected output response \( \{0, 0, 1\} \). Consequently, 110 is returned as a valid expected trace for \( p \).

Next, simulating \( C_e \) with the input vector \( \{10, 01, 10\} \) and the expected value \( \{1, 1, 0\} \) at \( p \), the value of \( s^2 \) and \( s^3 \) can be determined; that is, \( s^2 = 1 \) and \( s^3 = 1 \). Then, the truth table of \( p \) states that \( p \) evaluates to 1 under minterms (in terms of \( \{a, b, s\} \) \( \{100, 011\} \) and to 0 under \( \{101\} \). Note that this truth table is partially specified, since only the input and state explored by the test vectors are considered in the table. Therefore, \( \text{on}(p) = \{100, 011\} \) and \( \text{off}(p) = \{101\} \). The aSPFD of \( p \) contains two edges: \( (100, 101) \) and \( (011, 101) \).
A special case needs to be considered in Step 1. Because $\eta_{err}$ in each timeframe is not constrained, the SAT solver can assign any value to each $\eta_{err}$ to satisfy the formula. However, there is still a relationship between each $\eta_{err}$. For any two timeframes of the same test vector, $T_i$ and $T_j$, if the values of the primary input at these two timeframes are equal (i.e., $X^i = X^j$), and the value of the state $S^i$ and $S^j$ are the same, then the value of $\eta^i_{err}$ equals the value of $\eta^j_{err}$. This is obvious since the same values are input into the combinational circuitry in both timeframes. Hence, additional clauses are added to $\Phi$ to enforce this constraint. The required additional constraints are explained in the next example.

Example 3.5 With respect to Example 3.4, one can verify the assignment $100$ to $(p^1, p^2, p^3)$ can also satisfy the given output response. However, in this case, the value in terms of $\{a, b, s\}$ at $T_1$ and $T_3$ are both $100$, but $p^1$ and $p^3$ have opposite values. This is not a valid expected trace since it can cause a conflicting assignment in the truth table of $p$. Hence, additional constraints are required to prevent such an assignment being returned by SAT solvers.
Since the primary inputs \((a, b)\) have the value \((1, 0)\) at \(T_1\) and \(T_3\), \(p^1\) and \(p^3\) must be assigned with the same value if \(s^1\) and \(s^3\) have the same value. The following clauses are added to the SAT instance to enforce this constraint:

\[
(s^1 + s^3 + r) \cdot (\overline{s^1} + \overline{s^3} + r) \cdot (\overline{r} + p^3 + p^1) \cdot (r + p^1 + p^3)
\]

where \(r\) is a new variable that is 1 if \(s^1\) equals \(s^3\) (the first two clauses). The last two clauses ensure that \(p^1\) and \(p^3\) have the same value if \(s^1\) equals \(s^3\) (i.e., \(r = 1\)).

### 3.4.3 Optimizing \(a\)SPFDs with Don’t Cares

Sets of \(a\)SPFDs generated from the procedure described above may contain minterm pairs that are not required to be distinguished. This is because the procedure does not take into account all the don’t cares in the design. If a minterm \(m_i\) is a don’t care, all SPFD edges connected to \(m_i\) can be removed from the \(a\)SPFDs. Therefore, identifying don’t cares for \(\eta_{err}\) can further reduce the size of \(a\)SPFDs (i.e., the number of minterm pairs to be distinguished), and increase the number of qualified solutions when constructing the transformation.

#### 3.4.3.1 Combinational Don’t Cares

As discussed in Section 2.3 for combinational circuits, there are two types of combinational don’t cares: Satisfiability Don’t Cares (SDCs) and Observability Don’t Cares (ODCs). Since \(a\)SPFDs of nodes in combinational designs are built over the primary input space, only ODCs need to be considered. Recall that ODCs are minterm conditions where the value of the node has no effect on the values observed at the primary outputs. Hence, ODCs of \(\eta_{err}\) can only be found under \(V_c\). Minterms encountered under the simulation of \(V_c\) cannot be ODCs, because, otherwise, no erroneous behavior can be observed at the primary outputs. To identify ODCs, one can simply simulate the circuit with \(V_c\) and force \(\eta_{err}\) with a value complemented to the original simulation value, \(\overline{V_c}(\eta_{err})\). If no discrepancy is observed at the primary outputs, the associated minterm is an ODC and can be ignored.

Combinational ODCs of node \(\eta\) in a sequential design are assignments to the primary inputs and current states such that a change of values at \(\eta\) cannot be observed at the primary outputs.
or at the next states. Unlike in the case of combinational designs, combinational ODCs of sequential designs may be found in erroneous vector sequences as well. This is because the sequential design behaves correctly until the error is excited. Having this in mind, the following procedures can be added after Step 2 in Section 3.4.2 to obtain combinational ODCs.

Step 2a. Let $E_{T_1}$ denote the expected trace obtained in Step 2 in Section 3.4.2 and $\hat{S}$ denote the values of states in each timeframe. Another expected trace $E_{T_2}$ can be obtained by solving the SAT instance $\Phi$ again with additional constraints that (a) force $\hat{S}$ on all state variables so that the new trace can result in the same state transition, and (b) block $E_{T_1}$ from being selected as a solution again. Consequently, the expected trace obtained by solving the new SAT instance consists of different values at the transformation node such that the same state transition is generated.

Step 2b. Let $E_{T_2}$ be the second expected trace, and $T_i$ be the timeframe where $E_{T_1}$ and $E_{T_2}$ have different values. It can be concluded that the minterm at $T_i$ is a combinational don’t care, since, in this timeframe, the values of the primary outputs and the next states remain the same regardless of the value of $\eta$.

Step 2c. Repeat this procedure until no new expected trace can be found.

**Example 3.6** In Example 3.4, an expected trace, $E_T = 110$, has been obtained, and the state value is $\{s^1, s^2, s^3\} = \{0, 1, 1\}$. To obtain another expected trace at $p$, the following constraints are added to the SAT instance:

\[
(s^1)(s^2)(s^3) \quad (3.5) \\
(p^1 + p^2 + p^3) \quad (3.6)
\]

Equation 3.5 ensures states $\{s^1, s^2, s^3\}$ to have the value $\{0, 1, 1\}$, while Equation 3.6 prevents the SAT solver to assign $\{1, 1, 0\}$ to $\{p^1, p^2, p^3\}$ again. In this example, another expected trace of $p$, $E_{T_2} = 010$, can be obtained. Comparing $E_T$ and $E_{T_2}$, one can see that the two expected traces are different at $T_1$. This implies that the minterm 100 in terms of $\{a, b, s\}$ is a don’t care, since the values of $o^1$ and $s^2$ w.r.t. the input 100 remain the same regardless of the value of $p^1$. Hence, the aSPFD of $p$ can be reduced to contain only one edge, $(011, 101)$. 


3.4.3.2 Sequential Don’t cares

In addition to combinational don’t cares, sequential circuits can have sequential don’t cares that relate to the states of the circuits. The proposed procedure for constructing aSPFDs for sequential circuits has taken some sequential don’t cares into account. There are two types of don’t cares that this construction takes care of. The first one is equivalent states. Two states of a sequential circuit are equivalent if, for any input vector, they have identical values at the primary outputs and the corresponding next states are equivalent. Equivalent states are considered by the algorithm implicitly. When generating the expected trace, there are no constraints on state values. Hence, the SAT solver can assign values to \( \eta_{err} \) to cause any state transition as long as the expected output responses are complied. The SAT solver can freely explore equivalent states from the state transition relation. The second state don’t care is an unreachable state which has no incoming transitions from any state, including itself. Unreachable states are considered by construction. Since aSPFDs are constructed by simulating the design, the algorithm only considers the states that are reached during simulation. Hence, un-reachable states are excluded from aSPFDs automatically.

3.4.3.3 Conflicts in Multiple Expected Traces

In the case of sequential circuits, for the given input sequences and the expected output responses, there can exist multiple expected traces that satisfy the SAT instance in Step 1. The procedure described earlier for obtaining ODCs in sequential circuits identifies expected traces with the same state transitions. To further explore equivalent states, one can obtain a trace with different state transitions. This can be done by adding additional constraints to block \( \hat{S} \) assigned to state variables and solving the SAT instance again. However, these additional traces may assign different logic values to the transformation node for the same minterms and cause conflicts in the new truth table of the transformation node. The cases where such a conflict can arise are explained in the following discussion.

Let \( E_{T_1} \) and \( E_{T_2} \) represent two expected traces of the same node for the same test vector sequence. Assume a conflict occurs for minterm \( m \) between the assignment to \( E_{T_1} \) at cycle \( T_i \) and the assignment to \( E_{T_2} \) at cycle \( T_j \). In this instance, one of the two cases below is true:
• Case 1: The output responses and the states at cycle $T_{i+1}$ for $E_{T1}$ and $T_{j+1}$ for $E_{T2}$ are the same. This implies that the value of the transformation node under $m$ does not affect the output response or the next state. Hence, $m$ is a combinational ODC.

• Case 2: The next states are different. This can happen when the circuit has multiple state transition paths with the same initial transitions. Figure 3.8 shows an example of this scenario. Let $\eta$ be the transformation node. The graph depicts a state transition diagram for a single-output design. The diagram is simplified such that it depends on the value of $\eta$ only. The condition for a state transition occurrence is labelled on the edge and the value of the output is indicated below the state. Assume a test vector makes the design start at $S_0$. It takes at least three cycles to differentiate the transition $Path_a$ and $Path_b$, since the value of the primary output is not changed until the design is in $S_4$. Because the proposed analysis is bounded by the length of the input vector sequences, it may not process enough cycles to differentiate these different paths. Hence, multiple assignments at the transformation node can be valid within the bounded cycle range and cause conflicts when generating the new truth table of the transformation node. In the example, if the circuit is only unrolled for two cycles, both paths ($S_0 \rightarrow S_1$ and $S_0 \rightarrow S_2$) would seem to be the same from the observation of the primary outputs. Consequently, $\eta$ can have either logic 0 and logic 1 in $S_0$. Since the algorithm does not have enough information to distinguish the correct assignment, the minterm $m$ in this case is considered to be a don’t care as well. This issue can be resolved if vector sequences that are long...
enough are used instead.

As evident from the discussion above, it is desirable to consider all expected traces returned by the SAT solver to generate the truth table for the transformation. Step 1 and 2 in the algorithm in Section 3.4.2 are repeated until all the solutions to the SAT instance are obtained. When updating the truth table of the transformation node, minterms that have conflicting values in different expected traces are considered as don’t cares.

### 3.4.4 Validating $\alpha$SPFDs

For combinational circuits, $\alpha$SPFDs only explore the portion of the input space covered by the given input vectors. Similarly, for sequential circuits, $\alpha$SPFDs only consider states that are reachable during simulation of the given input vector sequences. As a result, $\alpha$SPFDs require less computation and memory resources. However, these benefits come with the penalty that the design has to undergo verification after restructuring to guarantee its correctness. This is because the transformation is guaranteed to be valid only under the input space exercised by the given set of input test vectors. In some cases, such as in rewiring, a full blown verification may not be required, but a faster proof method can be used [12,18,79].

Furthermore, because of approximation, the accuracy of the transformation depends on the amount of information provided by the input vectors. With more vectors, $\alpha$SPFDs become better representations of the original SPFDs. As a result, the chance that the modified circuits pass verification is higher. At the same time, the algorithm requires more resources to solve the problem. Hence, there is a trade-off between resolution and performance. Experiments show that this trade-off is a favorable one as, on average, in 90% of the cases the first transformation returned also passes verification with 1000 test vectors used for combinational designs and 500 vector sequences that are 10 cycles long used for sequential designs.

### 3.5 Logic Transformations with $\alpha$SPFDs

In this section, the procedure to systematically perform logic restructuring with $\alpha$SPFDs is presented. Let circuit $C_s$ be the specification and $C_e$ be the circuit under restructuring. The
Chapter 3. Automated Logic Restructuring with αSPFDs

Algorithm 3.1 Transformation using αSPFDs

1: \( C_e := \) Erroneous circuit

2: \( V := \) A set of input vectors

3: \( \eta_{err} := \) Transformation node

4: **procedure** Transformation\_With\_αSPFD\((C_e, V, \eta_{err})\)

5: \[ \mathcal{N} := \{ \eta_k \mid \eta_k \in C_e \text{ and } \eta_k \not\in TFO(\eta_{err}) \} \]

6: Compute αSPFDs of \( \eta_{err} \) and \( \mathcal{N} \) with \( V \) as discussed in Section 3.4

7: \[ E \leftarrow R_{\eta_{err}}^{app} \cup \bigcup_{i=1}^{k} R_{\eta_i}^{app} \text{ where } \{ \eta_1, \cdots, \eta_k \} \text{ are fanins of } \eta_{err} \text{ in } C_e \]

8: \( \text{Cover} \leftarrow \text{SelectCover}(\mathcal{N}) \)

9: Re-implementing \( \eta_{err} \) with the original fanins and the nodes in \( \text{Cover} \)

10: **end procedure**

goal is to re-implement the local network at \( \eta_{err} \) such that \( C_e \) implements the same function as \( C_c \). The proposed restructuring procedure uses αSPFDs to seek transformations at \( \eta_{err} \) that are constructed with one or more additional fanins.

The procedure is summarized in Algorithm 3.1. The basic idea is to find a set of nets such that the union of their αSPFDs is the superset of the αSPFD of the new transformation implemented at \( \eta_{err} \), as stated in Equation 2.8. Hence, the procedure starts by constructing the αSPFD of \( \eta_{err} \), \( R_{\eta_{err}}^{app} \), and those of all nets not in the transitive fanout cones of \( \eta_{err} \), \( TFO(\eta_{err}) \). Note that nets in \( TFO(\eta_{err}) \) cannot be used because they can generate a combinational loop. To minimize the distortion that may be caused by the transformation, the original fanins are kept for restructuring. This also reduces the number of SPFD edges that need to be covered. In other words, it is sufficient that αSPFDs of additional fanins only need to cover edges in \( R_{\eta_{err}}^{app} \) that have not been covered by αSPFDs of any original fanins (line 7). The function SelectCover is used to select a set of nodes (\( \text{Cover} \)) from nodes not in \( TFO(\eta_{err}) \) such that each uncovered edge is covered by at least one node in \( \text{Cover} \) (line 8). The function SelectCover is further discussed in the next subsections. Finally, a new two-level AND-OR network is constructed at \( \eta_{err} \) using the nodes in \( \text{Cover} \) as additional fanins according to Property 2.1 in Section 2.3.4.

Example 3.7 Returning to Example 3.3, the αSPFDs of nodes \( z_{mod}, b, d \) and \( f \) are shown in
Figure 3.9: aSPFDs of nodes in Figure 3.5

(a) aSPFD of $z_{mod}$

(b) aSPFD of $b$

(c) aSPFD of $d$

(d) aSPFD of $f$

Figure 3.10: Example circuit in Figure 3.5(a) after logic transformation

Figure 3.9 As shown in the figure, the edge $(110, 100)$ in the aSPFD of $z_{mod}$ (the dotted line) is the only SPFD edge that is not covered by any aSPFDs of the fanin of $z_{mod}$, \{f, d\}. Hence, the aSPFDs of additional fanins required for restructuring at $z_{mod}$ must contain this edge. In this example, the edge is contained in the aSPFD of $b$, as shown in Figure 3.9(b). Hence, $b$ can be used as the additional fanin for restructuring $z_{mod}$.

The minterm 100 is the only minterm in the onset of new $z_{mod}$ w.r.t. $V$. It implies $b = 0$, $d = 0$ and $f = 1$. Therefore, the new function of $z_{mod}$ is $\text{AND}(\overline{b}, \overline{d}, \overline{f})$. Using this new function, the $\text{XOR}$ gate can be removed, reducing the original gate count from eight to seven. The final circuit is shown in Figure 3.10.

Two approaches to find Cover are presented in the following subsections. The first is an optimal SAT-based approach in terms of minimizing the required support of fanin wires. The
second is a greedy approach that exchanges optimality for performance.

### 3.5.1 SAT-based Searching Algorithm

To implement the search in Algorithm 3.1 (line 8), the problem is formulated as an instance of Boolean satisfiability. Recall that the algorithm looks for a set of nodes outside the transitive fanout cone of \( \eta_{err} \) such that the union of aSPFDs of those nodes covers SPFD edges of \( R_{\eta_{err}}^{appx} \) that do not exist in the aSPFD of any fanins of \( \eta_{err} \).

**Lemma 3.1** The aSPFD of node \( \eta_k \), where \( \eta_k \in C_e \) and \( \eta_k \notin \{ TFO(\eta_{err}) \cup \eta_{err} \} \), is a subset of the SPFD of the corresponding node \( \tilde{\eta}_k \) in \( C_c \).

**Proof:** Suppose, towards a contradiction, the lemma is not true. Then there exists an edge \( e = (m_1, m_2) \) that belongs to the aSPFD of \( \eta_k \) in \( C_e \) and does not belong to the SPFD of \( \tilde{\eta}_k \) in \( C_c \). Thus, \( m_1 \) and \( m_2 \) both belong to either the onset or the offset of \( \tilde{\eta}_k \) in \( C_c \). Without loss of generality, assume they both belong to the onset. Since \( \eta_k \in C_e \) is identical to the corresponding node \( \tilde{\eta}_k \in C_c \), \( m_1 \) and \( m_2 \) have to produce the same values at the output of \( \eta_k \) as at the output of \( \tilde{\eta}_k \). Thus, \( m_1 \) and \( m_2 \) both belong to the onset of \( \eta_k \). Hence, by construction, \( (m_1, m_2) \) cannot belong to the aSPFD of \( \eta_k \). ■

**Lemma 3.2** With respect to a given vector set, \( V \), each edge in the aSPFD of node \( \eta_k \), where \( \eta_k \in C_e \) and \( \eta_k \notin \{ TFO(\eta_{err}) \cup \eta_{err} \} \), is contained in the aSPFDs of one or more of its fanins.

**Proof:** Assume the lemma is not true. Then there exists at least one minterm pair \( (m_1, m_2) \) in the aSPFD of \( \eta_k \) not belonging to the aSPFDs of the fanins in \( \eta_k \). By Lemma 3.1, the aSPFD of \( \eta_k \) in \( C_e \) is a subset of the SPFD of the corresponding node \( \tilde{\eta}_k \) in \( C_c \). Equation 2.8 states that each edge in the SPFD of \( \tilde{\eta}_k \) has to be contained in the SPFD of one or more of its fanins. Since \( \eta_k \) and its fanins implement the same function in \( C_c \) and \( C_e \), the SPFDs of \( \tilde{\eta}_k \) and its fanins in \( C_c \) are identical to the SPFD of the corresponding nodes in \( C_e \). Thus, \( (m_1, m_2) \) is contained in the SPFD of \( \eta_k \) and at least one of its fanins in \( C_e \). Hence, by construction, \( (m_1, m_2) \) has to belong to the aSPFDs of one of the fanins of \( \eta_k \). ■
In other words, Lemma 3.2 states that a node cannot have a value transition if none of its fanin has a value transition. This observation is used to constrain the solution space of the SAT instance of the searching problem.

Construction of the SAT instance is fairly straightforward. Each uncovered SPFD edge in the aSPFD of $\eta_{err}$ has a list of nodes whose aSPFD contain the edge. The SAT solver selects a node from each list such that at least one node in the list of each uncovered SPFD edge is selected. The set, $Cover$, consists of these selected nodes. The formulation of the SAT instance $\Phi$ is as follows. Each node $\eta_k$ is associated with a variable $w_k$. Node $\eta_k$ is added to the set $Cover$ if $w_k$ is assigned a logic value 1. The instance contains two components: $\Phi_C(W)$ and $\Phi_B(W, P)$, where $W = \{w_1, w_2, \cdots\}$ is the set of variables that are associated with nodes in the circuit and $P$ is a set of new variables introduced.

- **Covering clauses ($\Phi_C(W)$):** A covering clause lists the candidate nodes for an uncovered edge. A satisfied covering clause indicates that the associated edge is covered. One covering clause, $c_j$, is constructed for each uncovered edge $e_j$ in the aSPFD of $\eta_{err}$. Let $D$ be the candidate nodes whose aSPFDs contain edge $e_j$, clause $c_j$ contains $w_k$ if the aSPFD of $\eta_k$ covers $e_j$; that is,

$$\bigvee_{\eta_k \in D} w_k$$

Hence, this clause is satisfied if one of the included candidate nodes is selected.

- **Blocking clauses ($\Phi_B(W, P)$):** Blocking clauses define the condition where a candidate node $\eta_k$ should not be considered as a solution. They help to prune the solution space and prevent spending time on unnecessary searches. For each node $\eta_k \notin \{TFO(\eta_{err}) \cup \eta_{err}\}$, Lemma 3.2 implies that the aSPFD of $\eta_k$ does not cover more edges if all of its fanins are selected already. Hence, for each candidate node $\eta_k$, a new variable $p_k$ is introduced; $p_k$ is assigned a logic value 1 if all of the fanins of $\eta_k$ are selected, and 0 otherwise. Consequently, $w_k$ is assigned a logic value 0 (i.e., $\eta_k$ is not considered for the solution) when $p_k$ has a logic value 1. The blocking clause for node $\eta_k = f(\eta_1, \eta_2, \cdots, \eta_m)$, where $\eta_i, 1 \leq i \leq m$, is a fanin of $\eta_k$, is as follows:
Chapter 3. Automated Logic Restructuring with αSPFDs

\[(\bigvee_{i=1}^{m} \overline{w_i} + p_k) \cdot \bigwedge_{i=1}^{m} (w_i + \overline{p_k}) \cdot (\overline{p_k} + \overline{w_k}) \quad (3.8)\]

Given a satisfying assignment for \(\Phi\), a node \(\eta_k\) is added to the set \(\text{Cover}\) if \(w_k = 1\). The covering clauses ensure that \(\text{Cover}\) can cover all the edges in the αSPFD of \(\eta_{\text{err}}\). Blocking clauses reduce the possibility of the same set of edges being covered by multiple nodes in \(\text{Cover}\). If the function derived by the satisfying assignment from the set \(\text{Cover} = \{w_1, w_2, \ldots, w_n\}\) fails formal verification, then \((\overline{w_1} + \overline{w_2} + \cdots + \overline{w_n})\) is added as an additional blocking clause to \(\Phi\) and the SAT solver is invoked again to find another transformation.

**Example 3.8** With respect to Example 3.7, the SAT instance of the searching problem, \(\Phi = \Phi_C \cdot \Phi_B\), is constructed as follows. Since there is only one SPFD edge \((110, 100)\) in the αSPFD of \(z_{\text{mod}}\) that needs to be covered, \(\Phi_C\) consists of one clause only, which indicates candidate nodes of which function can distinguish the minterm pair. In this case, the candidate nodes of edge \((110, 100)\) are \(b\) and \(e\). Therefore, the formulation of \(\Phi_C\) is

\[\Phi_C = (b + e)\]

That is, this edge is covered if the SAT solver assigns 1 to either \(b\) or \(e\).

Next, considering node \(e\), as stated in Lemma 3.2, the number of minterm pairs that \(e\) can distinguish is fewer than the number of the collective minterm pairs that its fanins, \(b\) and \(c\), can distinguish. Hence, if \(b\) and \(c\) are already selected as additional fanins to the new structure, \(d\) does not provide the ability to distinguish more minterm pairs and can be removed from the candidate list. To formulate this idea in SAT, the blocking clauses are constructed as follows:

\[\Phi_B = (\overline{b} + \overline{c} + p_e) \cdot (b + \overline{p_e}) \cdot (c + \overline{p_e}) \cdot (\overline{p_e} + \overline{c})\]

where \(p_e\) is a new variable that has the value of 1 if both \(b\) and \(c\) are selected (i.e., both are assigned with 1). When \(p_e\) equals 1, \(e\) is forced to be 0; that is, \(e\) cannot be selected.

Note that in the above formulation because there are no constraints on the number of nodes that should be selected to cover the edges, the solution returned by the solver may not be optimal. In order to obtain the optimal solution, in experiments, SAT instances are solved with
a pseudo-Boolean constraint SAT solver [80] that returns a solution with the smallest number of nodes. The use of a pseudo-Boolean solver is not mandatory and any DPLL-based SAT solvers [34, 36] can be used instead. A way to achieve this is to encode the counter circuitry from [58] to count the number of selected nodes. Then, by enumerating values $N = 1, 2, \ldots$, the constraint enforces that no more than $N$ variables can be set to a logic value 1 simultaneously or $\Phi$ becomes unsatisfiable. Constraining the number $N$ in this manner, any DPLL-based SAT solver can find the minimum size of $Cover$.

### 3.5.2 Greedy Searching Algorithm

Although the SAT-based formulation can return the minimum set of fanins to re-synthesize $\eta_{err}$, experiments show that, at times, it may require excessive runtime. To improve the performance of the runtime, the following greedy approach to search solutions is proposed:

**Step 1.** Let $E$ be the set of SPFD edges in the aSPFD of $\eta_{err}$ that needs to be covered. For each edge $e \in E$, let $N_e$ be the set of nodes $\eta \not\in \{ \text{TFO}(\eta_{err}) \cup \eta_{err} \}$ whose aSPFD contains the edge. Sort $e \in E$ in descending order by the cardinality of $N_e$.

**Step 2.** Select the edge, $e_{min}$, with the smallest cardinality of $N_{e_{min}}$. This step ensures that the edge that can be covered with the least number of candidates is targeted first.

**Step 3.** Select $\eta_k$ from $N_{e_{min}}$ such that the aSPFD of $\eta_k$ covers the largest set of edges in $E$ and add $\eta_k$ to $Cover$.

**Step 4.** Remove edges that can be covered by $\eta_k$ from $E$. If $E$ is not empty, go back to Step 1 to select more nodes.

The solutions identified by the greedy approach may contain more wires than the minimum set. However, experiments indicate that the greedy approach can achieve results of a similar quality to the SAT-based approach in a more computationally efficient manner.
Chapter 3. Automated Logic Restructuring with αSPFDs

3.5.3 On the Quality of Test Vector Simulation for SPFD Approximation

In theory, the transformations returned using αSPFDs may not pass the verification since they are based on a subset of the complete input vector space. Nevertheless, experiments show that the success rate is very high and more than 90% of the first fix returned by the method passes verification. This implies that a small number of test vectors can provide sufficient information to generate sets of transformations that qualify. The reasons why the proposed αSPFD-based representation has such a high success rate are elaborated in the following discussion.

SPFDs (or αSPFDs) can be defined over any input spaces. In the discussion so far, all SPFDs are defined over the primary input space, which can be referred to as global SPFDs. On the other hand, local SPFDs refer to SPFDs that define over the immediate fanin space of a node. For simplicity, the term SPFD is used to refer to global SPFD. The discussion below shows that with a proper selection of input vectors, an αSPFD and SPFD of a node can be translated into the same local SPFD of the node. As a result, transformations constructed with the immediate fanin of the node based on this αSPFD and SPFD are the same.
Chapter 3. Automated Logic Restructuring with αSPFDs

Definition 3.3 The local minterms of a node \( \eta_k \) are minterms in terms of the immediate fanins of \( \eta_k \).

Note that because the functions of internal nodes are many-to-one functions (i.e., multiple inputs can result in the same output), many primary input minterms can be mapped to the same local minterms w.r.t. a node.

Definition 3.4 The local SPFD (or local αSPFD) of a node \( \eta_k \) specifies the local minterms in the onset of \( \eta_k \) that have to be distinguished from the local minterms in the offset of \( \eta_k \).

The local αSPFD of \( \eta_k \), \( R^{\text{appx}}_{\eta_k} = (V', E') \), can be translated from \( R^{\text{appx}}_{\eta_k} = (V, E) \) through the following steps. Let \( M_i \) be a primary input minterm and \( m_i \) be the corresponding local minterm of \( \eta_k \). First, for every \( M_i \in V \), \( m_i \) is added to \( V' \). Then, for every edge \( e = (M_i, M_j) \), where \( e \in E \), an edge between \( (m_i, m_j) \) is added to \( E' \).

Example 3.9 Consider the circuit shown in Figure 3.11(a) and its truth table shown next to it. Figure 3.11(c) depicts the SPFD of \( z \) (in terms of the primary inputs), while Figure 3.11(e) depicts the local SPFD of \( z \) (in terms of the immediate fanin, \( f, e \)). As shown in the truth table, the local minterm \( (d, e) = (1, 1) \) is a don’t care. Hence, it is not included in the local SPFD of \( z \). Notice that multiple primary input minterms can be mapped to the same local minterms. For instance, when \( (a, b, c) \) equals \((0, 0, 1)\) or \((1, 0, 1)\), \( (d, e) \) equals \((0, 1)\). Hence, if the αSPFD of \( z \) contains one of these two primary input minterms, the local αSPFD of \( z \) would include the local minterm \( 01 \). One can verify that the αSPFD of \( z \) shown in Figure 3.11(d) can be translated to a local αSPFD, which is exactly the same as the one in Figure 3.11(e).

The local network that can satisfy an SPFD (or an αSPFD) is synthesized by identifying local minterms in the onset of the node [51]. Hence, given an SPFD, \( R \), and an αSPFD, \( R^{\text{appx}} \), of the same node, the transformations constructed based on \( R \) or \( R^{\text{appx}} \) are the same if the same local SPFD can be derived from \( R \) and \( R^{\text{appx}} \). This can be achieved if the input vector set used to construct αSPFDs complies with the following lemma.

Lemma 3.3 Given an SPFD, \( R_{\eta_k} \), and an αSPFD, \( R^{\text{appx}}_{\eta_k} \), of node \( \eta_k \), and let \( R^{\text{loc}}_{\eta_k} \) be the local SPFD of \( \eta_k \) translated from \( R_{\eta_k} \) and \( R^{\text{appx}}_{\eta_k} \) be the local αSPFD of \( \eta_k \) translated from \( R^{\text{appx}}_{\eta_k} \).
Both $R^\text{loc}_{\eta_k}$ and $R^\text{appx}_{\eta_k}$ contain the same pairs of minterms to be distinguished if for each possible local minterm of $\eta_k$ that is not a don’t care, one of its corresponded primary input minterms is included in the $R^\text{appx}_{\eta_k}$.

**Proof:** To show that $R^\text{appx}_{\eta_k}$ and $R^\text{loc}_{\eta_k}$ contain the same pairs of minterms to be distinguished, it requires (a) that both $R^\text{appx}_{\eta_k}$ and $R^\text{loc}_{\eta_k}$ contain the same number of vertices and (b) that every edge in $R^\text{loc}_{\eta_k}$ is also in $R^\text{appx}_{\eta_k}$. The first requirement can be derived directly from the assumption that for each local minterm of $\eta_k$, one of its corresponding primary input minterms is included in $R^\text{appx}_{\eta_k}$. This leads to the conclusion that $R^\text{appx}_{\eta_k}$ contains the same number of $R^\text{loc}_{\eta_k}$. For the second requirement, toward a contradiction, assume that $R^\text{loc}_{\eta_k}$ contains an edge $(m_i, m_j)$, where $m_i$ and $m_j$ are local minterms of $\eta_k$, that is not in $R^\text{appx}_{\eta_k}$. Now, due to the assumption that $R^\text{appx}_{\eta_k}$ contains at least one primary input minterm that can be mapped to each local minterm of $\eta_k$, there must be a mapped edge $(M_i, M_j)$, where $M_i$ and $M_j$ are primary input minterms, in $R^\text{appx}_{\eta_k}$ for $(m_i, m_j)$. Since every minterm in $R^\text{appx}_{\eta_k}$ is mapped to a minterm in $R^\text{appx}_{\eta_k}$, the edge $(m_i, m_j)$ must exist in $R^\text{appx}_{\eta_k}$. Hence, by construction, an edge in $R^\text{loc}_{\eta_k}$ must be in $R^\text{appx}_{\eta_k}$ as well. ■

According to Lemma 3.3, if the set of primary input minterms exercised by the input vectors complies with the lemma, the $\alpha$SPFD constructed by the proposed approach contains all of the necessary information about the required function at the transformation node. As a result, the transformation that is constructed based on the $\alpha$SPFD can pass verification. Furthermore, if the transformation is constrained to have at most $N$ fanins, one may conclude that $2^N$ test vectors are sufficient to perform the restructuring if the selected test vectors comply to Lemma 3.3. Since $N$ is usually much less than the number of the primary inputs, $2^N$ test vectors are a small portion of the complete vector space.

Another observation about $\alpha$SPFDs is that, for some cases in sequential circuits, $\alpha$SPFDs may contain minterm pairs that are not required to be distinguished due to a lack of information from the input vectors. The following lemma shows that in these cases, there is no loss in the solution.

**Lemma 3.4** Given two $\alpha$SPFDs, $R^\text{appx}_i$ and $R^\text{appx}_j$, let $R^\text{appx}_i \subset R^\text{appx}_j$. Any functions that
satisfy $R_j^{appx}$ also satisfy $R_i^{appx}$.

**Proof:** Suppose, toward a contradiction, that this is not true. Then, there exists a function, $f$, that satisfies $R_j^{appx}$, but it does not satisfy $R_i^{appx}$. This implies that $f$ fails to distinguish at least one minterm pair required by $R_i^{appx}$. Because $R_i^{appx}$ is a subset of $R_j^{appx}$, such a minterm pair must exist in $R_j^{appx}$ as well. Thus, $f$ cannot satisfy $R_j^{appx}$. By contradiction, all functions that satisfy $R_j^{appx}$ have to satisfy $R_i^{appx}$. ■

### 3.6 Extension to Multiple Locations

All discussion so far has deal with restructuring only one location each time. However, as noted in the introduction, in practice there are several situations where restructuring at multiple locations is necessary. In this section, the procedure to apply the proposed methodology to perform multiple transformations in a combinational design is demonstrated. This concept can be easily extended to sequential circuits as well.

Given two combinational designs $C_c$ and $C_e$ and, without loss of generality, assume transformations need be applied on two locations, $\eta_{err1}$ and $\eta_{err2}$, in $C_e$ simultaneously such that $C_e$ becomes functionally equivalent to $C_c$. Depending on the locations of $\eta_{err1}$ and $\eta_{err2}$, as shown in Figure 3.12, a different approach can be followed.

The first case (Figure 3.12(a)) depicts the situation where the transitive fanout cones of $\eta_{err1}$ and $\eta_{err2}$ are exclusive to each other. This implies that these two locations can be restructured independently, because changes at one location do not affect the changes at the other. In this case, the proposed methodology discussed in Section 3.4.1 can be applied directly on each
location. Note that nodes in the transitive fanout cone of the transformation nodes that have not been restructured cannot be used to construct transformations, because the functionality of those transformation nodes is not finalized yet. For instance, assume that $\eta_{\text{err}1}$ is restructured first, all nodes in $TFO(\eta_{\text{err}1})$ and $TFO(\eta_{\text{err}2})$ are excluded from the candidate list for the searching procedure. However, after the completion of the restructuring of $\eta_{\text{err}1}$, nodes in $TFO(\eta_{\text{err}1})$ can be used as candidates for transformations at $\eta_{\text{err}2}$.

In the second case shown in Figure 3.12(b), there are common POs in the transitive fanout cone of the two nodes. The discrepancy observed at those POs may require fixes on both nodes to be resolved. Here, an approach, similar to the one described for sequential designs in Section 3.4.2, is summarized below:

Step 1. Extract the expected trace $E_T1$ for $\eta_{\text{err}1}$ and $E_T2$ for $\eta_{\text{err}2}$. This can be done by formulating a Boolean satisfiability instance from the circuit $C_e$ where $\eta_{\text{err}1}$ and $\eta_{\text{err}2}$ are marked as primary inputs.

Step 2. Obtain combinational don’t cares of $\eta_{\text{err}1}$ by solving all possible expected traces on $\eta_{\text{err}1}$ while the value of $\eta_{\text{err}2}$ is as specified in $E_T2$. Then, as in Section 3.4.3 if there are conflict assignments between expected traces, the corresponding minterms are don’t cares.

Step 3. Construct a partially specified truth table of $\eta_{\text{err}1}$ from the expected traces. The table specifies the function that can resolve all erroneous observations at POs when it is implemented at $\eta_{\text{err}1}$, and $\eta_{\text{err}2}$ is assigned with $E_T2$.

Step 4. Generate the $\alpha$SPFD of $\eta_{\text{err}1}$ from the truth table and construct a qualified transformation.

Step 5. Apply the transformation and repeat Step 2 to Step 4 for $\eta_{\text{err}2}$.

The procedures described above for two locations can be easily generalized when restructuring occurs in three or more places in a design. Overall, the procedure restructures one transformation node in each iteration from Step 2 to Step 4. This is necessary in order to obtain the correct combinational don’t cares for the transformation node, since they can be
different depending on the transformations applied at other nodes. Hence, after each transformation, the expected trace of the next transformation node is recalculated to take the effects of the previous transformations into account. As such, the order of which transformation node that is restructured first may result in a different final design.

Finally, the third case of the relation of two transformation locations is shown in Figure 3.12(b), where $\eta_{err2}$ is inside $TFO(\eta_{err1})$. In this case, the same procedure described for the second case can be applied. Although it may seem that $\eta_{err1}$ needs to be restructured before $\eta_{err2}$, it is not necessary. This is because aSPFDs are constructed based on the expected traces extracted in Step 1 of the procedure. Those values are what the transformation nodes should behave after restructuring. Therefore, even if $\eta_{err2}$ is restructured first, the minterm pairs that the fanins of $\eta_{err2}$ can distinguish after restructuring can be determined without constructing the transformation at $\eta_{err1}$. Consequently, $\eta_{err2}$ can be restructured before $\eta_{err1}$.

### 3.7 Experimental Results

The proposed logic restructuring methodology using aSPFDs is evaluated in this section. ISCAS’85 benchmarks are used for experiments on combinational transformations, while ISCAS’89 benchmarks are used for sequential cases. Those benchmarks are used because they are the common benchmarks used in other relevant work. The diagnosis algorithm from [58] is used to identify the restructuring locations and Minisat [35] is the underlying SAT solver. The restructuring potential of the aSPFD-based algorithms is compared with that of a logic correction tool from [75] which uses the dictionary-model of [29]. Both methodologies are compared against the results of error equation discussed in Section 3.3.1. The error equation check is implemented with the BDD package found in [81]. Experiments are conducted on a Core 2 Duo 2.4GHz processor with 4GB of memory while the runtime is reported in seconds.

#### 3.7.1 Experiment Setup

Table 3.3 summarizes the characteristics of benchmarks used in this experiment. Combinational benchmarks are listed in the first four columns, while sequential benchmarks are shown in the
last four columns. The table includes the number of primary inputs, the number of flip-flops and the total number of gates in each column, respectively.

In the experimental setup, three different complexities of modifications are injected in the original benchmark so that the effectiveness of the approach can be evaluated. Performance of the proposed methodology is evaluated with the ability to correct designs with those changes. The locations and the types of modifications are randomly selected. Simple complexity modifications (suffix “s”) involve the addition or deletion of a single wire, replacement of a fanin with another node, and a gate-type replacement. Moderate modifications (suffix “m”) on a gate include multiple aforementioned changes on a single gate. The final type of modification complexity, complex (suffix “c”), inject multiple simple complexity modifications on a gate and those in the fanout-free fanin cone of the gate.

For each of the above types, five testcases are generated from each benchmark. The proposed algorithm is set to find, at most, 10 transformations for each location identified first by the diagnosis algorithm. Functional verification is carried out at the end to check whether the 10 transformations are valid solutions.

### 3.7.2 Logic Restructuring of Combinational Designs

The first set of the experiments evaluates the proposed methodology for a single location in combinational circuits. Experimental results are summarized in Table 3.4. In this experiment,
Table 3.4: Combinational logic transformation results for various complexities of modifications

(a) Performance of the methodology

<table>
<thead>
<tr>
<th>Circ.</th>
<th>Error loc.</th>
<th>Error equat.</th>
<th>Dict. model</th>
<th>aSPFD</th>
<th>Avg time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1355_s</td>
<td>5.3</td>
<td>100%</td>
<td>19%</td>
<td>81%</td>
<td>3.5</td>
</tr>
<tr>
<td>c1908_s</td>
<td>18.0</td>
<td>84%</td>
<td>13%</td>
<td>84%</td>
<td>18.9</td>
</tr>
<tr>
<td>c2670_s</td>
<td>9.2</td>
<td>98%</td>
<td>11%</td>
<td>82%</td>
<td>21.9</td>
</tr>
<tr>
<td>c3540_s</td>
<td>7.2</td>
<td>100%</td>
<td>28%</td>
<td>86%</td>
<td>9.3</td>
</tr>
<tr>
<td>c5315_s</td>
<td>6.4</td>
<td>100%</td>
<td>25%</td>
<td>100%</td>
<td>7.6</td>
</tr>
<tr>
<td>c7552_s</td>
<td>11.8</td>
<td>88%</td>
<td>19%</td>
<td>50%</td>
<td>25.7</td>
</tr>
<tr>
<td>c1355_m</td>
<td>2.7</td>
<td>100%</td>
<td>13%</td>
<td>100%</td>
<td>32.0</td>
</tr>
<tr>
<td>c1908_m</td>
<td>5.8</td>
<td>100%</td>
<td>3%</td>
<td>83%</td>
<td>11.0</td>
</tr>
<tr>
<td>c2670_m</td>
<td>5.2</td>
<td>96%</td>
<td>4%</td>
<td>60%</td>
<td>95.4</td>
</tr>
<tr>
<td>c3540_m</td>
<td>3.2</td>
<td>100%</td>
<td>25%</td>
<td>100%</td>
<td>54.2</td>
</tr>
<tr>
<td>c5315_m</td>
<td>9.6</td>
<td>94%</td>
<td>2%</td>
<td>100%</td>
<td>46.7</td>
</tr>
<tr>
<td>c7552_m</td>
<td>8.8</td>
<td>100%</td>
<td>9%</td>
<td>91%</td>
<td>39.2</td>
</tr>
<tr>
<td>c1355_c</td>
<td>3.7</td>
<td>96%</td>
<td>0%</td>
<td>73%</td>
<td>38.4</td>
</tr>
<tr>
<td>c1908_c</td>
<td>15.8</td>
<td>47%</td>
<td>41%</td>
<td>70%</td>
<td>19.0</td>
</tr>
<tr>
<td>c2670_c</td>
<td>12.4</td>
<td>98%</td>
<td>31%</td>
<td>62%</td>
<td>33.2</td>
</tr>
<tr>
<td>c3540_c</td>
<td>3.0</td>
<td>100%</td>
<td>7%</td>
<td>67%</td>
<td>122.4</td>
</tr>
<tr>
<td>c5315_c</td>
<td>6.4</td>
<td>97%</td>
<td>16%</td>
<td>100%</td>
<td>20.0</td>
</tr>
<tr>
<td>c7552_c</td>
<td>20.6</td>
<td>64%</td>
<td>20%</td>
<td>50%</td>
<td>23.7</td>
</tr>
<tr>
<td>Average</td>
<td>8.6</td>
<td>93%</td>
<td>16%</td>
<td>80%</td>
<td>29.2</td>
</tr>
</tbody>
</table>

(b) Statistics of transformations

<table>
<thead>
<tr>
<th>Circ.</th>
<th>Avg # wires (greedy)</th>
<th>Min # wires (SAT)</th>
<th>Avg # corr/loc.</th>
<th>% verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1355_s</td>
<td>1.7</td>
<td>1.7</td>
<td>8.3</td>
<td>100%</td>
</tr>
<tr>
<td>c1908_s</td>
<td>1.4</td>
<td>1.4</td>
<td>8.1</td>
<td>90%</td>
</tr>
<tr>
<td>c2670_s</td>
<td>2.4</td>
<td>2.2</td>
<td>6.2</td>
<td>100%</td>
</tr>
<tr>
<td>c3540_s</td>
<td>1.1</td>
<td>1.1</td>
<td>4.5</td>
<td>100%</td>
</tr>
<tr>
<td>c5315_s</td>
<td>1.9</td>
<td>-</td>
<td>5.4</td>
<td>89%</td>
</tr>
<tr>
<td>c7552_s</td>
<td>1.7</td>
<td>-</td>
<td>3.1</td>
<td>88%</td>
</tr>
<tr>
<td>c1355_m</td>
<td>2.1</td>
<td>2.0</td>
<td>7.0</td>
<td>100%</td>
</tr>
<tr>
<td>c1908_m</td>
<td>2.5</td>
<td>2.5</td>
<td>5.6</td>
<td>100%</td>
</tr>
<tr>
<td>c2670_m</td>
<td>3.4</td>
<td>2.9</td>
<td>9.4</td>
<td>100%</td>
</tr>
<tr>
<td>c3540_m</td>
<td>1.6</td>
<td>1.6</td>
<td>6.1</td>
<td>84%</td>
</tr>
<tr>
<td>c5315_m</td>
<td>2.9</td>
<td>-</td>
<td>5.7</td>
<td>100%</td>
</tr>
<tr>
<td>c7552_m</td>
<td>1.9</td>
<td>-</td>
<td>6.9</td>
<td>100%</td>
</tr>
<tr>
<td>c1355_c</td>
<td>2.9</td>
<td>2.9</td>
<td>3.3</td>
<td>100%</td>
</tr>
<tr>
<td>c1908_c</td>
<td>1.4</td>
<td>1.3</td>
<td>7.2</td>
<td>100%</td>
</tr>
<tr>
<td>c2670_c</td>
<td>1.7</td>
<td>1.7</td>
<td>4.7</td>
<td>100%</td>
</tr>
<tr>
<td>c3540_c</td>
<td>3.6</td>
<td>3.4</td>
<td>3.8</td>
<td>100%</td>
</tr>
<tr>
<td>c5315_c</td>
<td>2.7</td>
<td>-</td>
<td>9.1</td>
<td>100%</td>
</tr>
<tr>
<td>c7552_c</td>
<td>1.9</td>
<td>-</td>
<td>3.5</td>
<td>91%</td>
</tr>
<tr>
<td>Average</td>
<td>2.0</td>
<td>-</td>
<td>6.4</td>
<td>96%</td>
</tr>
</tbody>
</table>
circuits are simulated with a set of 1000 input vectors that consists of a set of vectors with high stuck-at fault coverage and random-generated vectors. The high stuck-at fault coverage vectors are obtained from the authors in [82]. Table 3.4(a) compares the proposed methodology with the error equation approach and the dictionary model approach discussed in Section 3.2.1.

The first column lists the benchmarks and the types of modification inserted, as described in Section 3.7.1. The second column has the average number of locations returned by the diagnosis program for the five experiments. The percentage of those locations where the error equation approach proves the existence of a solution is shown in the third column. Note that the error equation approach is only used to check whether there exists a solution at the transformation node. As discussed in Section 3.3.1, the solution found by the error equation approach is a function in terms of the primary inputs. This may not be useful since this function can be complicated and fails to reuse existing design circuitry. The next two columns show the percentage of locations (out of those in the second column) for which the dictionary-approach and the proposed aSPFD approach can successfully find a valid solution. A valid solution is one in which the restructured circuit passes verification. The last column contains the average runtime, including the runtime of verification, to find all 10 transformations using the greedy heuristic.

Taking c1908_s as an example, there are, on average, 18 locations returned by the diagnosis program. The error equation check returns that 15 (84% of 18) out of those locations can be fixed by re-synthesizing the function of the location. The dictionary approach successfully identifies two locations (13% of 15) while the aSPFD approach can restructure 13 locations (84% of 15). This shows that the proposed approach is seven times more effective than the dictionary approach. Overall, the proposed methodology outperforms the dictionary approach in all cases and achieves greater improvement when the modification is complicated.

The quality of the transformations, in terms of the wires involved as well as some algorithm performance metrics, are summarized in Table 3.4(b). Here, only cases where a valid solution is identified by the proposed algorithm are considered. The second column lists the average number of additional wires returned by the greedy algorithm and the third column lists the minimum number of wires selected by the optimal SAT-based searching algorithm. As shown
in the table, the greedy heuristic performs well compared to the SAT-based approach. Because the SAT-based approach may run into runtime problems as the number of new wires increases, it times out ("-"") after 300 seconds if it does not return with a solution.

As mentioned earlier, the algorithm is set to find, at most, 10 transformations for each location. The fourth column of Table 3.4(b) shows the average number of transformations identified for each location. One can see that for all cases, more than one transformation can be identified. This is a desirable characteristic since engineers can have more options to select the best fit for the application. The final two columns show the percentage of transformations that pass verification. The first column of these two columns only considers the first identified transformation, while the second column has the percentage of all 10 transformations that pass verification. One can observe that the vast majority of first-returned transformations pass verification. Hence, based on Table 3.4(a) and (b), it can be concluded that \( \alpha \)SPFDs are useful for restructuring the design, and when a valid solution is found, it is often the first transformation identified. This observation confirms the viability of \( \alpha \)SPFDs.

The breakdown of the number of additional wires selected by the greedy approach (along with the original gate support) for a transformation that passes verification is depicted in Figure 3.13. It shows that more than half of the transformations are constructed with less than
Figure 3.14: Performance of restructuring with variable vector sizes for combinational designs

three additional wires. This suggests that the transformations only alter the design through small changes, which is important in logic rewiring, debugging or when applying engineering changes. Note that some cases do not require any additional wires; re-synthesis, according to the αSPFD at those transformation nodes, is sufficient to construct a transformation that can rectify the design.

Next, the performance of the restructuring with various numbers of test vectors is investigated. Four sizes are used: 250, 500, 1000 and 2000 test vectors. Test vectors consist of the same set of high stuck-at fault coverage vectors, plus various numbers of random vectors. The results are depicted in Figure 3.14. Figure 3.14(a) shows the percentage of the locations where the proposed algorithm can identify a valid transformation. As shown, the success rate is increased as the size of input vectors increases for each error complexity group. This is expected since more vectors provide more information for αSPFDs. The chance that the algorithm incorrectly characterizes a minterm as a don’t care is also reduced.

Although using a larger vector set can improve the success rate of the restructuring, it comes with the penalty that more computational resources are required to tackle the problem. The average runtime is plotted in Figure 3.14(b) and normalized by comparing it to the runtime of the case with 250 vectors. Each line represents one error complexity type. Taking Complex as an example, the runtime is 12 times longer when the vector size is increased from 250 to 2000. Note that there is a significant increase when the size of the vector set increases from 1000 to
Looking back to Figure 3.14(a), one may see that the success rate of cases when 1000 vectors are used is close to the success rate of those with 2000 vectors. This suggests that for those testcases, 1000 input vectors can be a good size to have a balance between the resolution of solutions and the runtime performance.

### 3.7.3 Logic Restructuring of Sequential Designs

The second set of the experiments evaluates the performance of logic restructuring with aSPFDs in sequential designs. The vector set for sequential circuits contains 500 random input vector sequences with a length of 10 cycles. To verify the correctness of transformations, a bounded sequential equivalence checker [83] is used. This tool verifies the resulting design against the reference within a finite number of cycles, which is set to 20 cycles in our experiment. The experimental results are summarized in Table 3.5.

The success rate of the proposed methodology for sequential circuits is recorded in Table 3.5(a). The first column of the table lists the benchmarks used in this set of the experiment. The second column presents the average number of locations for transformations reported by the diagnosis program while the percentage of these locations that are proven to be correctable by error equation are recorded in the third column. Because error equation is developed for combinational circuits, the sequential circuits are converted into combinational ones by treating the states as pseudo-inputs and pseudo-outputs. In this way, the number of locations reported by the error equation approach is the lower bound of the locations that are correctable. This is because the error equation approach is searching for a function at the transformation node such that the new design with the function implemented at the transformation node is combinationally functional equivalent to the reference design. That is, not only the primary outputs, but also the next states, have the same functionality as the reference design. This constraint discards any solutions that utilize equivalent states.

The percentage of locations in which the proposed methodology finds a valid transformation is reported in the fourth column. Overall, our approach can restructure 39% of the locations. The reason why the algorithm fails to correct some of the locations is because the input vector sequences do not provide enough information to generate a good aSPFD. This occurs when the
algorithm characterizes a minterm as a don’t care when this minterm is not exercised by the input vector sequences, or when conflict values are required for this minterm, as discussed in Section 3.4.3. Consequently, the resulting transformation does not distinguish all the necessary minterm pairs that are required to correct the design. The average of the overall runtime is reported in the last column.

Table 3.5(b) summarizes the statistics of identified transformations in the cases where a valid solution is identified. The second and the third columns report the average number of additional wires used in the transformations and the average number of transformations per location, respectively. As in the combinational cases, the transformation at some locations only needs to be re-synthesized with the existing fanin nets without any additional wires. This is the reason why cases such as s510 use less than one additional wire on average. The next two columns show the percentage of cases where the first transformation passes verification, and the percentage of 10 transformations that pass verification. Similar to the combinational circuits, there is a high percentage of the first transformations that pass verification if the proposed methodology returns a valid transformation. This indicates that aSPFD is a good metric to prune out invalid solutions.

Finally, these valid solutions are checked for whether or not they are unique to the sequential aSPFD-based algorithm. Note that a common way to deal with sequential designs is to convert them into combinational ones where state elements are treated as pseudo-primary inputs and pseudo-primary outputs. Subsequently, the design can be restructured as a combinational design. However, with this approach, transformations identified require maintaining equivalence of the primary outputs and the state elements between the restructured design and the reference design. To check whether the transformations identified by the proposed sequential algorithm can as well be identified by the simple combinational approach, combinational equivalence checking between the restructured design and the reference design is conducted. If two designs are not combinationally functional equivalent, it means that the transformation changes the state assignments as well. Such transformations will not returned by the combinational approach. Overall, 82% of the valid transformations are uniquely identified by the sequential approach and they cannot be found by the combinational logic restructuring method.
Table 3.5: Sequential logic transformation results for various complexities of modifications

(a) Performance of the methodology

<table>
<thead>
<tr>
<th>Circ.</th>
<th>Error loc.</th>
<th>Error equat.</th>
<th>aSPFD</th>
<th>Avg. time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s510_s</td>
<td>2.4</td>
<td>100%</td>
<td>75%</td>
<td>384</td>
</tr>
<tr>
<td>s713_s</td>
<td>5.0</td>
<td>72%</td>
<td>0%</td>
<td>325</td>
</tr>
<tr>
<td>s953_s</td>
<td>1.8</td>
<td>100%</td>
<td>33%</td>
<td>223</td>
</tr>
<tr>
<td>s1196_s</td>
<td>1.8</td>
<td>100%</td>
<td>56%</td>
<td>237</td>
</tr>
<tr>
<td>s1238_s</td>
<td>1.6</td>
<td>100%</td>
<td>38%</td>
<td>781</td>
</tr>
<tr>
<td>s1488_s</td>
<td>2.8</td>
<td>86%</td>
<td>43%</td>
<td>258</td>
</tr>
<tr>
<td>s510_m</td>
<td>2.0</td>
<td>100%</td>
<td>90%</td>
<td>68</td>
</tr>
<tr>
<td>s713_m</td>
<td>2.8</td>
<td>43%</td>
<td>36%</td>
<td>689</td>
</tr>
<tr>
<td>s953_m</td>
<td>1.6</td>
<td>63%</td>
<td>40%</td>
<td>105</td>
</tr>
<tr>
<td>s1196_m</td>
<td>1.2</td>
<td>83%</td>
<td>66%</td>
<td>27</td>
</tr>
<tr>
<td>s1238_m</td>
<td>2.6</td>
<td>85%</td>
<td>72%</td>
<td>218</td>
</tr>
<tr>
<td>s1488_m</td>
<td>3.4</td>
<td>100%</td>
<td>0%</td>
<td>83</td>
</tr>
<tr>
<td>s510_c</td>
<td>1.6</td>
<td>100%</td>
<td>38%</td>
<td>166</td>
</tr>
<tr>
<td>s713_c</td>
<td>3.4</td>
<td>71%</td>
<td>47%</td>
<td>1124</td>
</tr>
<tr>
<td>s953_c</td>
<td>2.2</td>
<td>73%</td>
<td>0%</td>
<td>122</td>
</tr>
<tr>
<td>s1196_c</td>
<td>2.0</td>
<td>50%</td>
<td>20%</td>
<td>588</td>
</tr>
<tr>
<td>s1238_c</td>
<td>1.2</td>
<td>100%</td>
<td>14%</td>
<td>328</td>
</tr>
<tr>
<td>s1488_c</td>
<td>1.8</td>
<td>71%</td>
<td>30%</td>
<td>98</td>
</tr>
<tr>
<td>Average</td>
<td>2.1</td>
<td>90%</td>
<td>39%</td>
<td>236</td>
</tr>
</tbody>
</table>

(b) Statistics of transformations

<table>
<thead>
<tr>
<th>Circ.</th>
<th>Avg # wires</th>
<th>Avg # corr/loc.</th>
<th>% verified</th>
<th>% unique</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>First</td>
<td>All</td>
</tr>
<tr>
<td>s510_s</td>
<td>0.3</td>
<td>1.8</td>
<td>100%</td>
<td>92%</td>
</tr>
<tr>
<td>s713_s</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>s953_s</td>
<td>1.0</td>
<td>3.3</td>
<td>100%</td>
<td>37%</td>
</tr>
<tr>
<td>s1196_s</td>
<td>2.0</td>
<td>5.0</td>
<td>83%</td>
<td>92%</td>
</tr>
<tr>
<td>s1238_s</td>
<td>1.1</td>
<td>5.0</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>s1488_s</td>
<td>1.7</td>
<td>5.0</td>
<td>83%</td>
<td>46%</td>
</tr>
<tr>
<td>s510_m</td>
<td>0.3</td>
<td>4.2</td>
<td>100%</td>
<td>38%</td>
</tr>
<tr>
<td>s713_m</td>
<td>0.6</td>
<td>1.4</td>
<td>100%</td>
<td>41%</td>
</tr>
<tr>
<td>s953_m</td>
<td>1.2</td>
<td>1.2</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>s1196_m</td>
<td>1.8</td>
<td>2.6</td>
<td>100%</td>
<td>72%</td>
</tr>
<tr>
<td>s1238_m</td>
<td>2.2</td>
<td>4.3</td>
<td>100%</td>
<td>76%</td>
</tr>
<tr>
<td>s1488_m</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>s510_c</td>
<td>0.5</td>
<td>1.5</td>
<td>100%</td>
<td>92%</td>
</tr>
<tr>
<td>s713_c</td>
<td>1.0</td>
<td>1.0</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>s953_c</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>s1196_c</td>
<td>0.5</td>
<td>2.3</td>
<td>50%</td>
<td>32%</td>
</tr>
<tr>
<td>s1238_c</td>
<td>0</td>
<td>–</td>
<td>100%</td>
<td>–</td>
</tr>
<tr>
<td>s1488_c</td>
<td>1.7</td>
<td>1.5</td>
<td>33%</td>
<td>27%</td>
</tr>
<tr>
<td>Average</td>
<td>1.0</td>
<td>3.1</td>
<td>92%</td>
<td>68%</td>
</tr>
</tbody>
</table>
Figure 3.15: Performance of restructuring with variable vector sizes for sequential designs
Next, the impact of test vector sizes on the performance of the presented methodology is studied. Here, the number of test vector sequences is set to 100, 200, 500 and 700. These test sequences have a length of 10 cycles and are randomly generated. The success rate and the normalized runtime are shown in Figure 3.15(a) and Figure 3.15(c) respectively. One can see that the behavior observed earlier for the combinational cases is also observed here. The success rate of the restructuring decreases as the number of the test sequences decreases. Among different error complexities, the benchmarks with complex errors are affected most. This is because a complex error can be excited in various ways and requires more test sequences to fully characterize the erroneous behavior. As a result, the algorithm needs more vector sequences to construct an accurate transformation. Figure 3.15(b) shows the percentage of the first transformation identified by the methodology that passes verification. One can see that cases with 100 input vector sequences have the lowest rate. This is because, with fewer vector sequences, the αSPFD contains fewer minterm pairs to be distinguished. As a result, there are more nets that qualify as part of the transformation. However, most of them do not pass verification. Overall, with 200 or more input vector sequences, the algorithm is able to find a transformation that passes verification by the first few iterations. Lastly, Figure 3.15(c) shows a significant reduction of the runtime with the decrease of the number of vector sequences.

### 3.7.4 Transforming Multiple Locations

The last set of experiments investigates the performance of the algorithm in simultaneous restructuring at multiple locations. A single simple error is inserted into two or three locations in the combinational benchmarks. Locations and errors are randomly selected. 1500 random vectors are used for each testcase. The procedure is configured to stop when a valid solution is identified or when, at most, 10 iterations have occurred and the set of transformations fails. These results are summarized in Table 3.6.

Table 3.6(a) summarizes the results of what occurs when restructuring is applied at two locations, while Table 3.6(b) contains a summary of the results regarding what occurs when three locations are restructured simultaneously. In both tables, the second column contains the average number of locations returned by the diagnosis program for the five experiments. Note
Table 3.6: Logic transformation at multiple locations

(a) two-location

<table>
<thead>
<tr>
<th>Circ.</th>
<th>Error loc.</th>
<th>Succ. rate</th>
<th>Avg # wires</th>
<th>Avg # trans. fail</th>
<th>Avg. time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>c880</td>
<td>9.2</td>
<td>39%</td>
<td>1.1</td>
<td>0</td>
<td>329</td>
</tr>
<tr>
<td>c1355</td>
<td>10.0</td>
<td>66%</td>
<td>1.2</td>
<td>0</td>
<td>649</td>
</tr>
<tr>
<td>c1908</td>
<td>10.0</td>
<td>36%</td>
<td>1.0</td>
<td>0</td>
<td>852</td>
</tr>
<tr>
<td>c2670</td>
<td>8.6</td>
<td>37%</td>
<td>1.1</td>
<td>0</td>
<td>1596</td>
</tr>
<tr>
<td>c3540</td>
<td>7.4</td>
<td>70%</td>
<td>0.9</td>
<td>0.03</td>
<td>2626</td>
</tr>
<tr>
<td>Average</td>
<td>9.0</td>
<td>50%</td>
<td>1.1</td>
<td>0.006</td>
<td>1210</td>
</tr>
</tbody>
</table>

(b) three-location

<table>
<thead>
<tr>
<th>Circ.</th>
<th>Error loc.</th>
<th>Succ. rate</th>
<th>Avg # wires</th>
<th>Avg # trans. fail</th>
<th>Avg. time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>c880</td>
<td>10.0</td>
<td>32%</td>
<td>1.6</td>
<td>0</td>
<td>666</td>
</tr>
<tr>
<td>c1355</td>
<td>10.0</td>
<td>68%</td>
<td>1.2</td>
<td>0.06</td>
<td>1226</td>
</tr>
<tr>
<td>c1908</td>
<td>10.0</td>
<td>16%</td>
<td>1.4</td>
<td>0</td>
<td>1136</td>
</tr>
<tr>
<td>c2670</td>
<td>9.6</td>
<td>32%</td>
<td>1.3</td>
<td>0</td>
<td>2608</td>
</tr>
<tr>
<td>c3540</td>
<td>9.2</td>
<td>48%</td>
<td>1.1</td>
<td>0</td>
<td>2757</td>
</tr>
<tr>
<td>Average</td>
<td>9.8</td>
<td>39%</td>
<td>1.3</td>
<td>0.01</td>
<td>1679</td>
</tr>
</tbody>
</table>

that because there are multiple errors, the diagnosis program may return numerous candidate solution tuples [75]. As such, in the experiments, at most, 10 location tuples are randomly picked as the candidates for restructuring.

The third column lists the percentage of the selected location tuples that our algorithm successfully identifies a valid transformation for each location in the tuple. The average number of additional wires required to construct the transformations are shown in the fourth column. For example, in the case of two-location restructuring for c1335, our algorithm is able to identify corrections for seven out of ten tuples returned by the diagnosis tool. The average number of additional wires used to construct the corrections is only 1.2. The empirical observation is that when the number of additional wires required increases, there is a high chance that the transformation will not pass verification. The fifth column shows, in the cases where a valid transformation is identified, the average number of transformation tuples that fail the verification before a valid one is found. As shown in the table, in all cases, valid solutions, if any, are usually identified at the beginning of the search process. This result is consistent with
the observation in the single-transformation experiments described earlier. Finally, the average runtime is recorded in the sixth column. As expected, the runtime increases as the number of restructured locations increases. In summary, it is seen that for both cases our technique corrects, on average, 50% and 39% of location tuples with less than two additional wires for each constructed transformation. The result confirms that the proposed methodology has the ability to restructure multiple locations efficiently.

3.8 Summary

In this chapter, a simulation-based procedure for a new representation of SPFDs, namely αSPFDs, is first presented. The αSPFD is an approximation of the original SPFD, as it only contains information that is explored by the simulation vectors. Next, an αSPFD-based logic restructuring algorithm for both combinational and sequential designs is presented. This technique can be used for a wide range of applications, such as logic optimization, debugging and applying engineer changes. The work is further extended to perform logic restructuring at multiple locations. Experiments demonstrate that αSPFDs provide a powerful approach to restructuring a logic design to a new set of specifications. This approach is able to construct required logic transformations algorithmically and restructure designs at a location where other methods fail.
Chapter 4

Soft Error Rate Reduction with $a$SPFDs

4.1 Introduction

As the complexity of VLSI designs increases, the demand for higher reliability also increases, especially for devices with high security needs or long-term usability. Among all reliability issues, soft errors, in particular, have become a serious problem due to reduced feature size and lower supply voltage [84]. Soft errors are produced when radiation particles pass through the semiconductor resulting in the generation of free electron-hole pairs. Semiconductor devices may be exposed to two types of radiation: external radiation and intrinsic radiation. The main source of external radiation is atmospheric cosmic rays, while the source of intrinsic radiation are $\alpha$-particles emitted from the packaging material of silicon chips.

When devices are hit with a high flux of radiation, the collected electrons generate a transient noise pulse if the induced charge exceeds the critical charge, $Q_{\text{crit}}$. $Q_{\text{crit}}$ is a physical design parameter related to parasitic capacitance and the logic voltage of transistors. This single-event transient (SET) can cause a functional error or data error, resulting in a single-event upset (SEU). The rate at which soft errors occur is referred to as soft error rate (SER).

Traditionally, the study of soft errors focuses on their impact on dynamic or static random-access memories. These memory arrays have high density structures and occupy a large portion
of the chip area, which makes them more vulnerable to radiation strikes. In contrast, soft errors in combinational logic are less critical; they would only affect the functionality of the design if a SET is captured by the registers or observed at the primary outputs. Past research has shown that combinational logic is much less susceptible to soft errors than memory elements [85], because of three masking mechanisms inherent in the combinational logic:

1. **Logic masking:** An SET is logically masked if it is blocked from propagating to any primary output or register because the value of a subsequent gate on the path is completely determined by the values of its unaffected fanins. For example, consider an AND gate, \( c = \text{AND}(a, b) \); propagating a transient error from \( a \) to \( c \) requires \( b \) to have a logic value 1.

2. **Electrical masking:** An SET is electrically masked if the amplitude of the SET is attenuated by subsequent logic gates due to the electrical properties of the gates such that it is too small to affect the circuit.

3. **Latching-window masking:** An SET which reaches a register, but is not captured due to the violation of the setup and hold time of the register is said to be masked.

These three mechanisms can significantly lower the impact of soft errors in combinational logic in comparison to memory circuits. However, due to device scaling and super-pipelining, the effectiveness of these mechanisms is diminished. For example, in a super-pipelined design, the depth of a logic network is decreased. Consequently, this reduces the probability of logic masking since it is easier to have a path consisting of gates whose values are changed in the presence of an SET. Higher clock frequencies increase the amount of latching per unit time by latches or registers. In turn, the chances that an SET is stored in latches or registers is increased. Hence, the opportunity for latching-window masking is reduced. As a result of increasing concern regarding the impact of soft errors on combinational logic, both mission-critical and mainstream commercial devices are required to be soft error tolerant.

To cope with soft errors, various physical level techniques have been developed to harden circuits from the impact of radiation [86–88]. The aim is to alter the physical structure of transistors in order to reduce the occurrence of SETs or to prevent SETs from propagating
through critical nodes to the primary outputs. While these methods are particularly effective in reducing the SER of a design, they are technology-dependent. That is, they rely on the information of the underlying technology after the cell library mapping stage. Moreover, all of the the above mentioned solutions treat the logic design as a black box; hence, the circuit implementation is unaltered in order to preserve both the design parameters and the logic optimization that have been achieved.

In contrast to established approaches, this dissertation presents a technology-independent, logic-level SER reduction methodology. This methodology minimizes the number of conditions where transient errors can affect the design functionality by rewiring wires through the use of $a$SPFDs, as presented in Chapter 3. It will be shown that SPFD-based rewiring not only optimizes the typical design objectives targeted by synthesis tools, but is also useful in minimizing the effect of transient errors. The proposed algorithm iteratively replaces a wire such that the cost function of the modified design is reduced. Two cost functions are considered: $OnlySER$, which aims at minimizing the SER regardless of the impact of rewiring on other design parameters, and $SERandAll$, which aims at reducing SER without degrading other design parameters. As a result of the technology-independent nature of this methodology, it is possible to consider SER as a design objective much earlier in the design cycle.

The remainder of the chapter is organized as follows. Section 4.2 summarizes the previous work on soft error analysis and reduction. Section 4.3 relates SPFDs to transient errors and demonstrates an approach to eliminate the condition for transient errors to affect the design functionality through the use of $a$SPFDs. The complete optimization algorithm is discussed in Section 4.4. Finally, the experiment that evaluates the proposed algorithm is presented in Section 4.5 followed by the summary in Section 4.6.

### 4.2 Preliminaries

Several SER evaluators have been proposed, such as SERA [84], FASER [89], and MARS-C [88]. These tools estimate the SER by accounting for the three masking mechanisms discussed previously: the probability of an SET propagating to a primary output or a register ($P_{\text{path}}$), the
rate of SET occurrence at a logic gate ($R_{\text{set}}$), and the probability of an SET arriving at a register during the latching-window ($P_{\text{latch}}$). $P_{\text{path}}$ is estimated by explicit enumeration of the input vector space in symbolic models [88] (e.g. BDDs), or by fault simulation on specific vectors [84]. $R_{\text{set}}$ is assessed using SPICE-based pre-characterization of the gate library. Finally, $P_{\text{latch}}$ is approximated based on the timing analysis information of flip-flops in the design [88]. In the experiment, SERA is used to compute the SER of designs and to evaluate the performance of the proposed methodology.

Techniques that mitigate soft errors via $R_{\text{set}}$ reduction are based on physical level design modifications. In [87, 88], gate resizing, which alters the W/L ratios of transistors in gates, is employed to increase the critical charge ($Q_{\text{crit}}$) of the gate. With the larger aspect ratio of a transistor, more charges need to be generated by a radiation event to result in a pulse with a magnitude larger than the switching threshold voltage of this transistor and to flip the logic value. A different approach is proposed in [86], which introduces extra transistors to the logic gate so that not only is the voltage margin increased to protect against the radiation noise, but the extra transistors also ensure that soft errors cannot propagate to subsequent gates. The authors in [86] redesign flip-flops to prevent the latching of SET inside the flip-flop itself. The drawback of these techniques is that potentially large costs in area and power are introduced in order to obtain significant reductions in SER.

In contrast to the aforementioned physical-level solutions, several techniques modify logic-level designs to increase logic masking opportunities by introducing functional redundancy. Classic techniques, such as triple modular redundancy (TMR), achieve this by duplicating the design. TMR replicates a design into three copies and the final result of the design is determined by the majority votes from the three copies. The obvious issue with these approaches is that they require a substantial increase in both area and power consumption. Almukhaizim et al. [24] add functional redundancy by adding selective redundant wires based on logic implications. Those implications include backward justification, direct implication and indirect implication. A similar approach proposed by Krishnaswamy et al. [90] computes signatures and observability don’t cares (ODC) of nodes in the design. The signature is a partial truth table of a Boolean function generated from the simulation of the design with random input vectors. By integrating
signatures and ODCs into synthesis techniques, partial redundancies among critical nodes of the design can be identified. Utilizing the redundancy information, a single-gate redundant logic is added to the fanin cone of the critical node to protect it from soft errors.

4.3 Transient Errors in Logic Circuits

To show how the SER in logic circuits can be reduced through the use of SPFDs, the relationship between the SPFD of a node and transient errors at the node that can affect the output of the design is first derived. Then, a procedure that employs SPFD-based rewiring to minimize the effect of transient errors in a circuit is demonstrated.

4.3.1 Relating SPFD to Transient Errors

A transient error that occurs at node $\eta_k$ is effective if it can change the design functionality. This requires a minterm at the primary inputs such that it sensitizes the error to the primary outputs. That is, the effect of the error propagates through a sequence of gates to a primary output. This requirement implies that the effect of transient errors at $\eta_k$ is related to the number of primary input minterms that can sensitize a value change at $\eta_k$ to the primary outputs.

Before discussing the relation of SPFDs and transient errors, the terms, minimum SPFD and care set, are defined below. They are used to derive the condition for transient errors to be effective in the later discussion.

**Definition 4.1** Let $F$ be the set of fanins of node $\eta$, and $R_\eta$ be the SPFD of $\eta$ that contains minterm pairs that the function at $\eta$ must distinguish. The minimum SPFD of fanin $\eta_k \in F$, denoted $R_{\eta_k \rightarrow \eta}^{\text{min}}$, contains minterm pairs $(m_i, m_j)$ in $R_\eta$ that can only be distinguished by $\eta_k$, but not by any node in $F - \eta_k$ [52].

**Definition 4.2** The care-set of an SPFD of a node consists of minterms that need to be distinguished by the function of the node.

In other words, the care-set of an SPFD of a node are minterms that are connected by edges in the SPFD. If the function of the node fails to distinguish those minterms, the design
functionality is altered.

**Example 4.1** Consider the circuit and its truth table shown in Figure 4.1(a) and Figure 4.1(b), respectively. Figure 4.1(c) depicts the SPFD of \( h \). Recall, each SPFD edge of a node must be included by one of the SPFDs of its fanins. The SPFDs of three fanins of \( h \) are shown in Figure 4.1(d) – (f) To construct the minimum SPFDs of these three fanins, each SPFD edge is checked for whether it can be distinguished by multiple fanins. Taking the SPFD edge \((000, 010)\) as an example, this edge is only distinguished by \( b \) (i.e., the edge exists in \( R_{b \rightarrow h} \)). Therefore, it is included in \( R_{b \rightarrow h}^{\text{min}} \) as shown in Figure 4.1(g). On the other hand, because the SPFD edge \((000, 101)\) is distinguished by \( e \) and \( f \), this edge is not included in the minimum SPFD of either fanin. The minimum SPFDs of the three fanin wires of \( h \) are shown in Figure 4.1(g) – (i).

Finally, as an example, the care-set of \( R_{b \rightarrow h}^{\text{min}} \) is \( \{000, 010, 101, 110, 111\} \).

The following lemmas state the relationship between a minimum SPFD of a node and conditions for a transient error at the same node to be effective. Such a condition is referred to as an effective condition.

**Lemma 4.1** Let \( R_{\eta_k \rightarrow \eta}^{\text{min}} \) be the minimum SPFD of node \( \eta_k \) to node \( \eta \), and \( M_{\eta_k} \) be the care-set of \( R_{\eta_k \rightarrow \eta}^{\text{min}} \). For each \( m_i \in M_{\eta_k} \), there exists a sensitized path from \( \eta_k \) to a primary output through \( \eta \).

**Proof:** According to Definition 4.1, \( M_{\eta_k} \) consists of minterms that the function at \( \eta_k \rightarrow \eta \), denoted \( f_{\eta_k \rightarrow \eta} \), must distinguish. Let \((m_i, m_j)\) be the pair of minterms that need to be distinguished by \( f_{\eta_k \rightarrow \eta} \), where \( m_i, m_j \in M_{\eta_k} \). Toward contradiction, assume that there does not exist any sensitized paths from \( \eta_k \) to any primary output through \( \eta \) under \( m_i \). It implies that \( f_{\eta_k \rightarrow \eta}(m_i) \) can be either 1 or 0, and the value does not affect the design functionality. That is, \( m_i \) is a don’t care to \( f_{\eta_k \rightarrow \eta} \) and cannot be included in \( M_{\eta_k} \). Hence, by construction, there exists a sensitized path from \( \eta_k \) to a primary output through \( \eta \) under \( m_i \). □

**Lemma 4.2** Let \( R_{\eta_k \rightarrow \eta}^{\text{min}} \) be the minimum SPFD of node \( \eta_k \) to node \( \eta \), and \( M_{\eta_k} \) be the care-set of \( R_{\eta_k \rightarrow \eta}^{\text{min}} \). A transient error that occurs at \( \eta_k \rightarrow \eta \) can affect the primary outputs if and only if under the condition where minterm \( m_i \in M_{\eta_k} \) is applied at the primary inputs.
Figure 4.1: Example of minimum SPFDs
Proof: The first part of the proof shows that, for each \( m_i \in \mathcal{M}_{\eta_k} \), if \( m_i \) is applied at the primary inputs, a transient error at \( \eta_k \) will affect the primary outputs. Since \( \mathcal{M}_{\eta_k} \) contains only the care-set of minterms, according to Lemma 4.1, there is a sensitized path from \( \eta_k \) through \( \eta \) to a primary output under these minterms. Hence, the effect of a flip at \( \eta_k \rightarrow \eta \) under \( m_i \in \mathcal{M}_{\eta_k} \) can be propagated and observed at the primary outputs. The second part of the proof shows that transient errors at \( \eta_k \rightarrow \eta \) have no effect if \( m_j \not\in \mathcal{M}_{\eta_k} \) is at the primary inputs. Again, since \( m_j \) is not in \( \mathcal{M}_{\eta_k} \), when \( m_j \) is applied at the primary inputs, none of the values of the primary outputs depends on the value of \( \eta_k \). Consequently, a flip at \( \eta_k \rightarrow \eta \) will not affect the design functionality. ■

Based on Lemma 4.2, the number of effective conditions at node \( \eta_k \), denoted as \( TEs(\eta_k) \), can be evaluated by the number of minterms in the care-set of \( R_{\eta_k}^{\min} \), denoted as \( \mathcal{M}_{\eta_k} \). That is,

\[
TEs(\eta_k) = |\mathcal{M}_{\eta_k}|
\]  

(4.1)

In other words, the number of effective conditions at the node can be determined by examining the number of distinct minterms in the \( R_{\eta_k}^{\min} \) of a node. Furthermore, the total number of effective conditions in circuit \( C \) equals to the sum of distinct minterms in the minimum SPFDs of all nodes in \( C \).

\[
TEs(C) = \sum_{\eta_i \in C} TEs(\eta_i)
\]  

(4.2)

**Example 4.2** Revisit the example circuit in Figure 4.1(a) and \( R_{\eta_k}^{\min} \) of immediate fanins of \( h \) (Figure 4.1(g) – (i)). Let \( \mathcal{M}_{i \rightarrow j} \) be the care-set of the minimum SPFD of the wire \( i \rightarrow j \). The care-set of each minimum SPFD of the three fanins is as follows:

\[
\mathcal{M}_{b \rightarrow h} = \{000, 010, 011, 110, 111\}
\]
\[
\mathcal{M}_{e \rightarrow h} = \{000, 100\}
\]
\[
\mathcal{M}_{f \rightarrow h} = \{000, 001\}
\]

According to Equation 4.1, the number of effective conditions for each fanin of \( h \) equals the number of minterms in the care-set of SPFD of each fanin. That is, \( TEs(b \rightarrow h) = 5 \),
Chapter 4. Soft Error Rate Reduction with \( a \)SPFDs

\[ TEs(e \rightarrow h) = 2, \text{ and } TEs(f \rightarrow h) = 2, \] respectively. Hence, there are nine effective conditions associated with the fanins of \( h \) in total where a transient error at these fanins can affect the primary output \( o_2 \).

As discussed earlier, the minterm pair \((000, 101)\) is not in any minimum SPFD of the three fanins of \( h \) because it can be distinguished by \( e \rightarrow h \) and \( f \rightarrow h \). As a result, the minterm 101 is not in the care-set of \( R_{e \rightarrow h}^{\text{min}} \) or \( R_{f \rightarrow h}^{\text{min}} \). According to Lemma 4.2 when a transient error occurs at \( e \rightarrow h \) or \( f \rightarrow h \) while 101 is applied to the primary input, the error has no effect on the design functionality. This property is the key idea explored in the proposed approach in order to eliminate the number of effective conditions in a circuit.

4.3.2 Minimization of Transient Errors

The previous section relates the number of effective conditions for transient errors at node \( \eta_k \) (\( TEs(\eta_k) \)) to be effective to the cardinality of the care-set of \( R_{\eta_k \rightarrow \eta}^{\text{min}} (M_{\eta_k \eta}) \). This implies that the susceptibility of a node to transient errors can be reduced if the size of the care-set of the minimum SPFD of a node is reduced. In this section, an approach to eliminate minterms in the care-set of a minimum SPFD by using SPFD-based rewiring techniques is demonstrated.

As shown in Example 4.1 a pair of minterms is eliminated from the \( R_{\eta_k \rightarrow \eta}^{\text{min}} \) of the fanins of a node if the pair is distinguished by at least two fanins to that node. In other words, the fewer minterm pairs that are uniquely distinguished by the fanins of a node, the fewer transient errors at the fanins of the node that can affect the design functionality. Therefore, the idea proposed is to increase the commonality between SPFDs of the fanins of a node by replacing the fanins.

Let \( F \) be the set of fanins of \( \eta \). Given two fanin nodes of \( \eta, \{\eta_s, \eta_t\}, \) where \( \eta_s, \eta_t \in F \), the cardinality of \( M_{\eta_s \rightarrow \eta} \) can be reduced if fanin \( \eta_t \) is replaced with another node, \( \eta_{\text{new}} \), such that \( \eta_{\text{new}} \rightarrow \eta \) distinguishes more minterm pairs in \( R_{\eta_s \rightarrow \eta} \) than \( \eta_t \rightarrow \eta \) does. In other words, \( R_{\eta_{\text{new}} \rightarrow \eta} \) shares more common SPFD edges with \( R_{\eta_s \rightarrow \eta} \). Candidates of the replacement node \( \eta_{\text{new}} \) can be identified according to Property 2.3 of SPFDs described in Section 2.3.4. That is, the SPFD of the new node needs to be the superset of the SPFD of the replaced node.

Example 4.3 Continuing from Example 4.2, the fanin \( b \rightarrow h \) is associated with the highest

\[ \]
Chapter 4. Soft Error Rate Reduction with aSPFDs

number of effective conditions ($T_{Es}(b \rightarrow h) = 5$). To reduce the number, one can replace the other two fanins of $h$, namely $e$ and $f$, with a new node, $p$, such that the connection, $p \rightarrow h$, also covers some of the SPFD edges in $R_{b \rightarrow h}^{\min}$ (Figure 4.1(g)). According to Property 2.3, $f \rightarrow h$ can be replaced with $c \rightarrow h$, since $R_{c \rightarrow h}$ (Figure 4.2(e)) is the superset of $R_{f \rightarrow h}$ (Figure 4.1(f)).

The modified circuit is shown in Figure 4.2(a), where the dotted line is the new connection. One can see that $c$ can distinguish two additional minterm pairs in $R_{b \rightarrow h}^{\min}$, $(000, 011)$ and $(000, 111)$, over the ones distinguished by $f \rightarrow h$. The new minimum SPFDs of $h$ and its fanins are shown in Figure 4.2(c) - (e). As a result, $T_{Es}(b \rightarrow h)$ is reduced from five to three, and the total
number of effective conditions at the fanins of h is reduced from nine to seven.

4.4 Optimization Algorithm

The complete algorithm is presented in Algorithm 4.1. It iteratively selects a rewiring operation that produces a better circuit based on a cost function. Two cost functions are used to evaluate the effect of the replacement: OnlySER and SERandAll. The first cost function, OnlySER, aims to minimize the soft error rate regardless of the impact on other design parameters, such as area, delay, power and fault coverage. Let $improve(x)$ be a function that returns the ratio between design parameter $x$ of the initial circuit over the same design parameter of the circuit after the wire replacement. OnlySER can be expressed as:

$$OnlySER = \max \{improve(SER)\}$$ \hspace{1cm} (4.3)

In contrast, the second cost function, SERandAll, reduces the SER as long as all the design parameters of the modified circuit are better or equal to those of the initial circuit. The function is defined as:

$$SERandAll = \max \{improve(SER)\},$$

Subject to:

$improve(Area) > 0$

$improve(Delay) \geq 0$

$improve(Power) \geq 0$

$improve(Faultcoverage) \geq 0$ \hspace{1cm} (4.4)

The optimization algorithm works as follows. In each iteration, it first uses aSPFDs, as discussed in Chapter 3, to compute $R^{min}$ for each wire in the circuit. Then, wires are sorted in descending order by the number of effective conditions (line 5 and 6). Then, the algorithm goes into the first for-loop which selects a wire as the target for optimization in that sorted order (line 8 – 21).
Chapter 4. Soft Error Rate Reduction with aSPFDs

Algorithm 4.1 Greedy algorithm for reducing SER using SPFD-based rewiring

1: \( C := \) Design to be optimized

2: \textbf{procedure} \textsc{Reduce_TEs USING SPFD}(C)

3: \hspace{1em} \textbf{repeat}

4: \hspace{2em} \textit{CF}_{old} := \text{Evaluate the cost function of } C

5: \hspace{2em} \text{Compute } \textit{R}_{\text{min}} \text{of all wires}

6: \hspace{2em} \textit{W} := \text{Sort wires in descending order of } TEs()

7: \hspace{2em} \textit{CF}_{\text{best}} := \textit{CF}_{old}

8: \hspace{2em} \textbf{for } w_i \in \textit{W} \text{ do}

9: \hspace{3em} \text{Perform rewiring to reduce } |M_{w_i}| \text{ and generate } k \text{ designs } \{C_1, \ldots, C_k\}

10: \hspace{3em} \textbf{for } C_j, 1 \leq j \leq k \text{ do}

11: \hspace{4em} \textit{CF}_j := \text{Evaluate the cost function of } C_j

12: \hspace{4em} \textbf{if } \textit{CF}_j > \textit{CF}_{\text{best}} \text{ then}

13: \hspace{5em} \textit{CF}_{\text{best}} := \textit{CF}_j

14: \hspace{5em} C_{\text{best}} := C_j

15: \hspace{4em} \textbf{end if}

16: \hspace{3em} \textbf{end for}

17: \hspace{2em} \textbf{if } \textit{CF}_{\text{best}} \neq \textit{CF}_{old} \text{ then}

18: \hspace{3em} C := C_{\text{best}}

19: \hspace{3em} \textbf{break}

20: \hspace{2em} \textbf{end if}

21: \hspace{2em} \textbf{end for}

22: \hspace{2em} \textbf{until} all wires in \( C \) has been tried

23: \textbf{end procedure}
### Table 4.1: Characteristics of benchmarks

<table>
<thead>
<tr>
<th>Circ.</th>
<th># PI</th>
<th># PO</th>
<th># Gates</th>
<th>Circ.</th>
<th># PI</th>
<th># PO</th>
<th># Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>b01</td>
<td>7</td>
<td>7</td>
<td>51</td>
<td>s298</td>
<td>17</td>
<td>20</td>
<td>119</td>
</tr>
<tr>
<td>b02</td>
<td>5</td>
<td>5</td>
<td>27</td>
<td>s382</td>
<td>24</td>
<td>27</td>
<td>158</td>
</tr>
<tr>
<td>b03</td>
<td>34</td>
<td>34</td>
<td>153</td>
<td>s344</td>
<td>24</td>
<td>26</td>
<td>160</td>
</tr>
<tr>
<td>b06</td>
<td>11</td>
<td>15</td>
<td>55</td>
<td>s349</td>
<td>26</td>
<td>26</td>
<td>161</td>
</tr>
<tr>
<td>b08</td>
<td>30</td>
<td>25</td>
<td>171</td>
<td>s526</td>
<td>24</td>
<td>27</td>
<td>173</td>
</tr>
<tr>
<td>b10</td>
<td>28</td>
<td>23</td>
<td>180</td>
<td>s444</td>
<td>24</td>
<td>27</td>
<td>181</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s510</td>
<td>25</td>
<td>13</td>
<td>211</td>
</tr>
</tbody>
</table>

For each target wire $w_i$, wire replacements that can reduce the cardinality of the care-set of $R_{w_i}^{\text{min}}$ as discussed in Section 4.3.2 are identified (line 9). It is possible to have multiple qualified replacements and each replacement generates a different design. Hence, the cost function of designs is evaluated to pick the best modification (line 10 – 16). The design with the best improvement compared to the original design is kept and the minimum SPFDs of each wire are recomputed for another iteration of optimization. This process is repeated until all the wires have been tried and there is no improvement to the cost function.

### 4.5 Experimental Results

The proposed approach is evaluated with benchmarks from ISCAS’89 and ITC’99. Only the combinational circuitry of those benchmarks is used. That is, registers are treated as pseudo-primary inputs and outputs of the design. All benchmarks used in the experiments are listed in Table 4.1. The first and fifth columns have the name of the benchmark, followed by the number of primary inputs, the number of primary outputs and the total gate counts.

In these experiments, the TSMC 0.13$\mu$m process and Synopsys Design Compiler are used to compute the area, critical path delay and dynamic power of the circuit. TetraMax is used to perform Automatic Test Pattern Generation (ATPG) and compute any loss in fault coverage due to rewiring. SERA [84] is used to compute the SER for each primary output of the design.
Table 4.2: Experimental results using the cost function (OnlySER)

<table>
<thead>
<tr>
<th>Circ</th>
<th>SER</th>
<th>Area</th>
<th>Delay</th>
<th>Power</th>
<th>Fault coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>b01</td>
<td>-17.1%</td>
<td>-4.2%</td>
<td>-20.0%</td>
<td>-2.4%</td>
<td>0.0%</td>
</tr>
<tr>
<td>b02</td>
<td>-10.8%</td>
<td>-4.6%</td>
<td>36.8%</td>
<td>-0.3%</td>
<td>-3.6%</td>
</tr>
<tr>
<td>b03</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>b06</td>
<td>-15.9%</td>
<td>-19.1%</td>
<td>45.2%</td>
<td>-12.5%</td>
<td>-2.3%</td>
</tr>
<tr>
<td>b08</td>
<td>-2.4%</td>
<td>-0.9%</td>
<td>17.2%</td>
<td>0.1%</td>
<td>-0.7%</td>
</tr>
<tr>
<td>b10</td>
<td>-5.3%</td>
<td>-1.9%</td>
<td>21.8%</td>
<td>-3.4%</td>
<td>-1.5%</td>
</tr>
<tr>
<td>s298</td>
<td>-17.5%</td>
<td>-6.5%</td>
<td>37.5%</td>
<td>-7.9%</td>
<td>-2.5%</td>
</tr>
<tr>
<td>s382</td>
<td>-25.1%</td>
<td>-11.2%</td>
<td>11.1%</td>
<td>-19.0%</td>
<td>-2.6%</td>
</tr>
<tr>
<td>s344</td>
<td>-7.4%</td>
<td>-9.9%</td>
<td>10.6%</td>
<td>-10.5%</td>
<td>-0.1%</td>
</tr>
<tr>
<td>s349</td>
<td>-7.3%</td>
<td>-6.9%</td>
<td>10.6%</td>
<td>-11.2%</td>
<td>-1.7%</td>
</tr>
<tr>
<td>s526</td>
<td>-16.7%</td>
<td>-5.1%</td>
<td>22.1%</td>
<td>-9.1%</td>
<td>-5.6%</td>
</tr>
<tr>
<td>s444</td>
<td>-12.0%</td>
<td>-2.7%</td>
<td>20.3%</td>
<td>-5.0%</td>
<td>-3.2%</td>
</tr>
<tr>
<td>s510</td>
<td>-16.3%</td>
<td>-16.7%</td>
<td>3.5%</td>
<td>6.2%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

Average: -11.8% -6.9% 16.7% -5.8% -1.8%

Experimental results of the proposed optimization approach with the OnlySER and SERandAll cost functions are summarized in Tables 4.2 and 4.3, respectively. Five parameters of a design are reported in the tables: soft error rate, area overhead, delay overhead, power overhead, and fault coverage.

In the first set of experiments, the only objective of the optimization is to reduce the SER of designs. As shown in Table 4.2, the proposed approach can reduce the SER of most designs substantially, with an average reduction of 11.8%. In particular, the SER of 6 out of 13 benchmarks is reduced more than 15%. Since the search is driven solely by the objective of reducing the SER, the impact of the proposed optimization on other design parameters can be significant. Overall, the area and power overhead of modified designs are improved. This is because after rewiring, some gates become dangling and can be removed. Consequently, the dynamic power consumption is reduced as well. Note that all benchmarks have been optimized in area before the SER reduction process is carried out. Since synthesis tools tend to optimize the design globally, it is possible to achieve a significant area reduction when local optimization like the one proposed is performed. However, critical path delay is increased significantly in many cases after the modification. For example, the delay of b02, b06, and s298 increases by more than 30%. The reason is that rewiring a new node can require a longer routing or increase the circuit level. Both can result in a longer path delay. A similar phenomenon of
changes in area, power, delay after rewiring is also observed in the experiments of other rewiring techniques [12, 74]. Finally, fault coverage is lost after the optimization. This is because the technique used in the proposed approach is to introduce redundancy to the design to increase the chance of logic masking.

In the second set of experiments, the impact of the SER optimization on other design parameters is monitored. As shown in Table 4.3, these additional constraints diminish the ability to reduce SER. However, the optimization achieved under such constraints often results in significant reductions in one or more of these design parameters. Taking s298 as an example, in addition to 20% of reduction in SER, area, delay and power overhead are reduced by 15%, 8% and 16%, respectively. Such reductions are not achieved when the algorithm optimizes SER only. Note that the fault coverage of s349 and s444 increases after the rewiring. As mentioned earlier, some gates can become dangling and removed from the design after rewiring. Hence, any redundant faults associated with those gates are removed as well. Consequently, the coverage fault of the new design is improved. However, these cases are exceptional, since in this experiment setup rewiring operation that lower fault coverage would not be selected.

Table 4.3: Experimental results using the cost function (SERandAll)

<table>
<thead>
<tr>
<th>Circ.</th>
<th>SER</th>
<th>Area</th>
<th>Delay</th>
<th>Power</th>
<th>Fault coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>b01</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>b02</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>b03</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>b06</td>
<td>-6.5%</td>
<td>-3.5%</td>
<td>0.0%</td>
<td>-5.6%</td>
<td>0.0%</td>
</tr>
<tr>
<td>b08</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>b10</td>
<td>-2.0%</td>
<td>-4.2%</td>
<td>-1.3%</td>
<td>-4.7%</td>
<td>0.0%</td>
</tr>
<tr>
<td>s298</td>
<td>-20.2%</td>
<td>-15.9%</td>
<td>-8.2%</td>
<td>-16.6%</td>
<td>0.0%</td>
</tr>
<tr>
<td>s382</td>
<td>-3.3%</td>
<td>-1.9%</td>
<td>-2.7%</td>
<td>-3.6%</td>
<td>0.0%</td>
</tr>
<tr>
<td>s344</td>
<td>-7.2%</td>
<td>-8.8%</td>
<td>-2.9%</td>
<td>-7.9%</td>
<td>0.0%</td>
</tr>
<tr>
<td>s349</td>
<td>-3.1%</td>
<td>-3.5%</td>
<td>0.0%</td>
<td>-3.4%</td>
<td>0.5%</td>
</tr>
<tr>
<td>s526</td>
<td>-1.1%</td>
<td>0.0%</td>
<td>-2.1%</td>
<td>-0.4%</td>
<td>0.0%</td>
</tr>
<tr>
<td>s444</td>
<td>-2.6%</td>
<td>-2.4%</td>
<td>-2.2%</td>
<td>-3.1%</td>
<td>0.9%</td>
</tr>
<tr>
<td>s510</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Average</td>
<td>-3.5%</td>
<td>-3.1%</td>
<td>-1.5%</td>
<td>-3.5%</td>
<td>0.1%</td>
</tr>
</tbody>
</table>
4.6 Summary

This chapter demonstrates the relationship between SPFDs and the number of transient errors that can affect the design functionality. A greedy algorithm is proposed to reduce the SER of a design by utilizing SPFDs. By implementing functionally-equivalent yet structurally-different transformations at each node of a design, the SER can often be reduced at no additional overhead to other design parameters. The proposed technique can be integrated as a part of the synthesis process, and makes it possible to consider optimization for reliability in an earlier stage of the VLSI design flow.
Chapter 5

Automated Software Solutions to
Silicon Debug

5.1 Introduction

Developing modern integrated circuits consists of several synthesis stages before a silicon prototype is fabricated. To ensure that each synthesis step does not introduce any errors (e.g., timing, functional, power), a corresponding verification stage is carried out to validate the design. During the pre-silicon process, engineers test devices in a virtual environment with sophisticated simulation [91], emulation [92] and formal verification [40, 93] tools to check the correctness of the RTL model against its functional specification. However, due to the growing complexity of functionality and the size of designs, it becomes infeasible to achieve 100% verification coverage within the strict time-to-market constraints. As such, functional bugs may escape pre-silicon verification and only be discovered during in-system silicon validation, where the design is exercised at speed. In addition, parasitic and inadequate process variation models may contribute errors during the fabrication. For example, because of the narrow feature size nowadays, a small deviation on the gate width of a transistor can significantly affect its threshold voltage. Consequently, silicon prototypes are rarely bug-free. In fact, more than 60% of design tape-outs require a re-spin, and more than half of these re-spins are due to logical or functional errors not discovered by pre-silicon verification [27]. Each re-spin dramatically increases the project
cost and the time-to-market. Therefore, it is important to develop a silicon debug flow that provides a short turn-around time when a silicon prototype fails.

A typical silicon debug process consists of several iterative sessions, as shown in Figure 5.1. Each iteration, referred to as one debug session, can be divided into two stages: data acquisition and data analysis. A debug session starts with data acquisition. In the data acquisition stage, test engineers set up the test environment to obtain the appropriate data from the chip under test while it is operated in real-time. Unlike pre-silicon verification, where values of internal signals can be easily obtained through simulation, observability of those for silicon debug is restricted in both space and time. Only signals that are connected to output pins can be probed, an insufficient scenario for debugging. For this reason, several Design-for-Debug (DfD) hardware components, such as scan chains or trace buffers, are used to access the internal signals. Nevertheless, adding new hardware comes with a cost. As such, the amount of acquired data is limited by the amount of additional DfD hardware integrated. These limits greatly inhibit accurate and effective debugging analysis.

During data analysis, the sparse amount of data acquired during the test is analyzed to prune the error candidates and to set up the data acquisition environment for the next debug session. This time-consuming and labor-intensive cycle continues until the root cause of the failures is determined. Several techniques have been proposed to automate the data analysis process that determine the root cause of failures solely based on the acquired data [94, 95]. Clearly, the quality of the data analysis is affected by the acquired data and the analysis can be effective if the data contains useful information regarding the error location. Hence, software solutions for silicon debug need to have the ability to identify sets of signals, as well as cycles during the execution, that are important and helpful in narrowing down the suspects. Furthermore, this set of signals should be concise because only a small proportion can be acquired during each silicon execution.

This dissertation proposes an automated software-based debug methodology that complements current data acquisition hardware solutions. The proposed methodology automates the data analysis step in Figure 5.1 to aid the engineer in discovering the root cause of the chip failure. This methodology identifies the potential locations of the error in a hierarchical manner,
and estimates the time interval where the error is excited. This allows the engineer to concentrate the manual investigation on a smaller set of locations within a more concise window of cycles. As a result, the amount of manual work performed by the engineer can be reduced. In addition, our methodology helps refine the debug experiment by setting up the data acquisition environment for the next debug session. It utilizes UNSAT cores to identify registers that may contain useful information, which can narrow down the suspect candidates. The new data acquired is fed to the subsequent automated data analysis cycle to eventually determine the root cause.

In practice, not all registers can be accessed by the data acquisition hardware. To comply with these hardware constraints and make the proposed methodology more practical, a search algorithm is presented to find alternatives for registers of interest that are not traceable by the hardware. The algorithm takes the hardware constraints into consideration and finds alternative states among the pre-selected registers such that the values of those registers may imply the untraceable ones. The proposed search algorithm is memory efficient because only a small window of the complete silicon trace is analyzed. Finally, because only one traceable register
group can be traced in each debug session, a simple ranking system is suggested to prioritize the traceable register groups according to the results from the proposed analysis.

Experiments on OpenCores and ISCAS’89 circuits are conducted. Results show that our methodology successfully determines the locations of the error and it also specifies the time interval in which the error is excited. Even with the hardware constraints considered, the methodology reduces, on average, 30% of the number of suspects that the engineer needs to investigate with only 8% to 20% of registers traced. To the best of our knowledge, this is the first study that presents a comprehensive analysis of the data acquired with modern in-silicon hardware like trace buffers and scan chains to reduce the number of iterations during silicon debug.

The remainder of the chapter is organized as follows. Section 5.2 summarizes prior work done on hardware and software solutions for silicon debug, as well as the background material. Section 5.3 discusses the challenges found in silicon debug compared to RTL debugging. Section 5.4 presents the proposed software solution to silicon debug, while Section 5.5 illustrates the searching algorithm for selecting alternatives for non-traceable registers. The ranking system is presented in Section 5.6. The experimental results and summary are given in Sections 5.8 and 5.9 respectively.

5.2 Preliminaries

As mentioned earlier, the main difficulty with silicon debug is the lack of access to the internal signals. In this section, two data acquisition hardware components used to enhance the observability of internal signals in chips are discussed. The data analysis algorithm and hardware test tools for identifying the root cause of failures are also reviewed.

5.2.1 Design for Debug Hardware Solutions

The behavior of internal signals in the chip can be observed only if the signals are routed to external pins. Since there are limited numbers of available pins on the chip, this approach may not provide sufficient information to perform debugging. To improve data acquisition from
silicon chips, two main ad-hoc DfD solutions are used in practice: scan chains and trace buffers.

5.2.1.1 Scan Chains

Scan chains are widely employed as a Design-for-Test (DfT) technique during manufacture test and are reused for silicon debug [96]. They provide a means to take a snapshot of the state of internal signals at a specific cycle.

A sample scan chain structure is depicted in Figure 5.2(a) In the normal operation mode, registers are connected in parallel to the combinational logic. When the design is switched to the test mode, all scanned registers are connected together into a serial shift register (the dotted line). This forms a serial scan chain. If all the registers in a design are converted to scannable elements, the architecture is known as full scan. If some non-scanned registers are left in the design, the architecture is known as partial scan. In the test mode, the values stored in scanned registers can be serially shifted out with a slower shift clock. This operation is referred to as scan dump. To perform this operation, the functional clocks of the design are halted. Consequently, scan dump cannot be operated in real time during silicon debug.
Since functional bugs may occur thousands of cycles after the initial start-up, it is desirable to resume the execution after scan dump. However, this is not possible if regular scan registers are used because the values stored in these registers are destroyed after the scan dump. To resume the execution from the same point, the environment needs to be reset and restarted from the initial setup. Non-destructive registers, on the other hand, consist of additional storage elements to preserve the stored values after the scan dump. As a result, it is possible to resume the execution at the same point after the design is halted. The downside of these types of scan cells is the area overhead and the additional load on the data input signal. In either case, a new state capture cannot occur until the previous scan dump has been completed. Consequently, it is not practical to acquire state values for several consecutive cycles using scan chains [97].

5.2.1.2 Trace Buffers

Trace buffers are another DfD technique that is complementary to scan chains [98, 99]. They allow data acquiring in real time by recording internal signals in an on-chip memory. As shown in Figure 5.2(b) a trace buffer contains control logic, called trigger logic, employed for the on-line monitoring of circuit behavior. Once the trigger condition is asserted, the on-chip memory can start/stop recording the logic values of the selected signals. Subsequently, the recorded data can be read via a low-bandwidth interface, such as a boundary scan. The advantage of trace buffers is that the values of signals can be collected for consecutive cycles, while scan chains only provide the value in a specific cycle. However, the drawback is that the amount of data is limited by the size of the embedded memory. Typical sizes of trace buffers range from 8Kb to 256Kb. The amount of data that can be acquired is constrained by the depth and the width of the trace buffer. The former limits the number of samples to be stored, while the latter limits the number of signals sampled in each clock cycle.

Trace buffers provide flexibility in what to observe in the chip during debugging. However, the number of signals that can be sampled is usually very limited. Only a set of pre-selected signals can be traced. These pre-selected signals are divided into groups and connected to the on-chip memory through a multiplexer, as shown in Figure 5.2(b). During execution, only one group can be selected and traced at a time. The traceable signals are typically manually selected
by the designer. Recently, several algorithms have been developed to automate the selection process [100–103]. In those works, the authors analyze the topology of a circuit to calculate the ability of a particular signal to restore the remaining nodes in the circuit when the signal is monitored by the DfD hardware. In the end, those algorithms select a small set of signals such that their values have a higher chance of restoring a significant amount of untraceable states.

5.2.2 Related Work on Data Analysis

Although DfD hardware enhancement increases the observability of internal signals, there is a lack of techniques that automate the data analysis process on the acquired data. Recently, there has been an effort to develop methodologies to aid the engineer in this part of the silicon debug process as summarized in the following.

The method proposed by Caty et al. [94] first performs scan dumps to collect silicon state data over consecutive cycles ($T_{n-k}, \cdots T_n$). This data is compared with the golden reference to determine the failing, or unmatched, registers at each cycle. Next, starting from $T_n$, it conducts back-tracing from those unmatched registers to identify the fault propagation paths during the cycle. Back-tracing is a process that marks nodes that may propagate the error effect. The process starts at erroneous outputs or registers and traverses the logic netlist in reverse topological order. For a gate, an input node is marked if a value change at that signal can result in a value change at the output of the gate. It stops when primary inputs and registers are reached. Next, the algorithm prunes the traces with unmatched registers at $T_{n-1}$; only traces that reach these unmatched registers are retained. This procedure is repeated cycle-by-cycle until it reaches a cycle where there are no more unmatched registers. A suspect list that consists of all gates on the traces is generated. After the error propagation paths are identified, a forward-tracing is carried out to further narrow down the suspects. This forward-tracing excludes a suspect gate if there is a propagation path from the suspect to a passing register. Those gates cannot be suspects because, otherwise, a discrepancy should be observed at the passing register.

Yen et al. [104] propose a similar approach. Their methodology isolates the critical cycles using a binary search paradigm based on the comparison between the silicon data and the sim-
ulation results. A *critical cycle* is the first cycle in which the state elements show a discrepancy between the reference responses and the actual ones. After the critical cycle is determined, this method identifies all possible sources of the failure by back-tracing from the unmatched registers. Finally, it injects a faulty value for candidates identified by back-tracing in the golden model and simulates the golden design to observe the responses at the critical cycle. If the response matches the observation of the faulty chip, the suspect is qualified as the candidate in the final list of suspects.

Note that both the aforementioned approaches utilize simple path-tracing techniques to prune the potential suspects of the failure. Although those tracing techniques are efficient, they are conservative when determining the potential suspects. In order to guarantee that the actual error location is selected by tracing, those techniques include all the possible paths that an error effect can propagate. Consequently, the result may not provide enough insight regarding the errors. Furthermore, for these two methods to work, the complete golden reference has to be available. Such a pre-requisite is not always available. For instance, in the case of functional errors, there may not exist one-to-one signal mapping from all registers in the actual design to variables in the golden reference. In this case, only logic values for the registers that have a reference mapping in the golden model can be checked, which may deteriorate the final resolution.

In order to perform accurate debugging, it is important to obtain as much data about internal signals as possible. For instance, both approaches discussed above rely on scan dumps to obtain the state values in the faulty silicon during the test. However, as mentioned in Section 5.2.1, the scan dump operation is not efficient if state values for consecutive cycles are required. Hence, instead of obtaining those values directly from the silicon chip, the approach proposed in [95] uses formal methods and on-chip support logic to restore the state information backwards in time. The on-chip hardware calculates signatures and stores them in the trace buffer during the chip execution. Then, based on the signatures, a unique or a small set of possible predecessor states that lead to the crash state is calculated. The precision of calculated predecessor states depends on the signature function. The less ambiguous signatures are, the smaller the set of possible predecessor states that will be identified. Obviously, the main
disadvantage of this approach is the area overhead due to the additional hardware structures for signature generation. Also, it does not pinpoint the root cause of failures.

The methodology proposed in this chapter constructs problems as SAT instances and allows SAT solvers to make smarter decisions when selecting suspects. It identifies the location of the error, as well as the cycles when the error is excited. Moreover, the proposed methodology assists with the data acquisition hardware setup, which is not addressed by any of the aforementioned works.

5.2.3 Test Equipment for Silicon Debug

The software analysis methodologies described previously isolate the failing logic in the silicon chip. This may be enough to understand the cause of the failure if the root cause is due to design errors. However, for electrical issues, more information may be required to find the ultimate cause of the failure. This necessitates the use of other test equipment.

There is a variety of test equipment available for silicon debug. The most basic (yet expensive and necessary) piece of equipment is an IC tester. This collects the failure information at the chip’s pin level under different frequencies or voltages, and exposes the overall sensitivities of the design. More information about the internal nodes can be obtained by e-beam probes, which are often used to observe internal nodes on metal wires [105]. Other probing tools, such as the laser voltage probe (LVP) and time resolved emission (TRE) can measure the actual waveforms of internal signals with tens of picoseconds of resolution [106]. Finally, a focused ion beam (FIB) is a metal-editing tool used for bug investigation. It allows the engineer to edit silicon chips to verify the hypothesis of the cause of the failure.

5.3 Silicon Debug Versus RTL Debugging

As discussed in Section 1.3, in addition to physical defects introduced during manufacturing, a silicon prototype can fail tests due to functional errors in the RTL representation of the design that are not detected by pre-silicon verification. Hence, when targeting functional errors, both silicon debug and RTL debugging try to identify the source of errors in the design against the
functional specification. Even so, there are several fundamental differences between the two debugging processes.

First, the characteristics of error behavior can be different. RTL debugging is performed in a controlled simulation environment. The interaction between the design and external modules is controlled by the given test vectors. Hence, the erroneous behavior can be reproduced if the same set of test vectors is used. This type of behavior is referred to as deterministic behavior.

Errors in silicon prototypes have deterministic behavior if the circuit is debugged on the tester or on an application board where the inputs are controlled synchronously. However, errors in silicon prototypes can behave non-deterministically as well. The error can be triggered by an event that cannot be replicated deterministically, such as an interruption from peripherals, or the timing of refresh cycles for dynamic memories [107]. As a result, errors with non-deterministic behavior are excited with probability and increase the difficulty of debugging.

Second, traces used to carry out RTL debug are generated through simulation at RTL level. The verification engineer can probe an unlimited number of signals and clock cycles. In contrast, silicon debug observability is restricted by the design and the DfD techniques discussed in Section 5.2.1 are required to observe the internal signals.

Finally, due to the limitations of the memory and computation resources of simulators, test sequences used for RTL debugging usually range from one hundred to a few thousand cycles. However, in the case of silicon debug, because silicon chips are operated at-speed in the designed operation system, test sequences can be orders of magnitude longer compared to test sequences in RTL debugging.

As discussed so far, one can see that silicon debug is a complex and difficult problem. Therefore, the assumptions listed in the next subsection are used to make the problem more feasible to solve, while still reflecting realistic practical concerns.

5.3.1 Problem Assumptions

The methodologies presented in this dissertation are constructed under the following assumptions:

- The erroneous silicon behavior is deterministic. That is, it can be replicated with the same
Chapter 5. Automated Software Solutions to Silicon Debug

set of test vectors. This assumption is necessary to replicate experiments and to obtain the values of multiple state elements at different cycles. It is also the fundamental underlying assumption of a silicon debug environment, such as the one depicted in Figure 5.1.

- Scan chains and trace buffers (Section 5.2.1) are utilized to obtain the values of internal states. In this scenario, the design is fully scanned and trace buffers can be programmed to capture the value of specific state elements.

- Methodologies in the following discussion deal with functional errors (bugs) in the design. Examples of functional bugs that escape to silicon can be found in [26]. Malfunctions due to electrical and fabrication defects are not considered in this dissertation.

- The golden model, such as a high-level behavioral model, is available during debugging to provide the correct responses of the design. Note that although this behavioral model may not provide access to the data on every single net in the implementation, the important information on the data and address buses, as well as the essential control signals that steer the data through the data-path, can be monitored.

- All discrepancies are due to a single error present in the RTL representation. Since most test vectors target specific functionality of the design, it is realistic to conclude that a failing test vector is due to a single error [108].

5.4 Automated Data Analysis

The complete flow of the methodology is shown in Figure 5.3. The overview of the methodology is given here first and the details of the implementation will be further discussed in the following subsections. The proposed methodology has three main objectives: (1) to identify the suspect modules that contain the error, (2) to find the critical interval of the error, and (3) to find the registers that may contain useful information about the error. A critical interval is defined below.

**Definition 5.1** Given an erroneous sequential circuit $C_e$ and its golden reference $C_r$, let $S_{C}^{n}(V)$ denote the value of the registers of circuit $C$ at cycle $n$ of the test sequence $V$. The critical
Figure 5.3: A single debug analysis session

cycle, w.r.t. test vector sequence $V$, is the last cycle in the simulation of $V$, $T_i$, such that $S_{C_c}^{i-1}(V) = S_{C_c}^{i-1}(V)$ and $S_{C_c}^i(V) \neq S_{C_c}^i(V)$, and no discrepancy is observed at the primary outputs prior to $T_i$.

Note that, it is possible that there exists a cycle, $T_j$, where $T_j < T_i$, such that $S_{C_c}^{j-1}(V) = S_{C_c}^{j-1}(V)$ and $S_{C_c}^j(V) \neq S_{C_c}^j(V)$, but no discrepancy is observed at the primary outputs prior to $T_i$. In this case, $T_j$ is not considered as the critical cycle because the error effect is eliminated at $T_{j-1}$ before it can reach the primary outputs to be observed.

**Definition 5.2** Given a critical cycle, $T_c$, a **critical interval** of $T_c$ is a window of cycle $(T_m, T_n)$ such that $T_m \leq T_c \leq T_n$.

In other words, for a given test vector sequence, the critical cycle is the first cycle where the value of the registers of the erroneous design does not match the value of those registers in the golden reference. A critical interval is a window of cycles that contains the critical cycle.

The proposed data analysis flow performs debugging in three steps. Note that these three steps can also constitute three stand-alone procedures. First, it diagnoses the circuit in a hierarchical manner, as described in Section 2.4.2. In each iteration of diagnosis, only modules
Chapter 5. Automated Software Solutions to Silicon Debug

(a) Hierarchical design

(b) History of hierarchical diagnosis

Figure 5.4: Hierarchical diagnosis

in the level that is currently investigated are considered as candidates for the error source. It has been shown that it is effective to use the design hierarchy information when searching between different components of a design [62]. Unlike in [62] where the debugging algorithm iterates the procedure until the lowest hierarchical level is reached, the algorithm in the proposed flow would only expand at most \( n \) hierarchy levels from the level ended in the last session during each debug session. This is referred to as \( n \)-level hierarchical diagnosis. Take the example in Section 2.4.2. The figure is shown again in Figure 5.4 for convenience. Assume the data analysis flow implements a two-level hierarchical diagnosis process. In the first debug session, the flow begins with the hierarchical diagnosis that stops after two hierarchical levels (Lv1 and Lv 2) have been analyzed, and determines that Module \( C_1 \) is the suspect module. Then, the flow performs timeframe diagnosis and data acquisition setup analysis for \( C_1 \), and ends the first debug session. In the second debug session, hierarchical diagnosis continues the analysis from level three (e.g., the suspect candidates are \( C_a \) and \( C_b \)). Hierarchical diagnosis operates this way to provide coarse solutions at the beginning of the debugging process. This helps the engineer to have a rough idea about where the error might be and to determine what additional information should be obtained to further prune the suspect list. Then, the hierarchical diagnosis process in the next debug session can utilize the data and return a more concise list of suspects.

Next, following hierarchical diagnosis, timeframe diagnosis is carried out to find a more precise estimate for the window of clock cycles in which the error may be excited. This interval can further reduce the time interval where the design needs to be analyzed in the next debug
Chapter 5. Automated Software Solutions to Silicon Debug

Algorithm 5.1 Timeframe diagnosis

1: $M_{List} := \text{list of suspect modules}$

2: $k := \text{size of timeframe interval}$

3: $T_b(T_e) := \text{beginning(end) timeframe of the trace}$

4: procedure TimeframeDiagnosis($M_{List}, k, T_b, T_e$)

5: $TM_{List} := \text{the new list containing timeframe modules}$

6: $TM_{sol} := \text{the timeframe diagnosis solutions}$

7: for all $M \in M_{List}$ do

8: for $t = T_b$ to $T_e$ incremented by $k$ do

9: $TM_{new} \leftarrow \text{Construct a new timeframe module consists of } \{M^t \ldots M^{t+k}\}$

10: $TM_{List} \leftarrow TM_{List} \cup TM_{new}$

11: end for

12: end for

13: $TM_{sol} \leftarrow \text{Debug with candidates from } TM_{List}$

14: return $TM_{sol}$

15: end procedure

session. In addition, during the test, signals only need to be traced within the new reduced window.

Finally, if the root cause of the error cannot be determined with the data in hand, the engineer needs to determine what additional data should be obtained and set up the capture environment for the next debug session. Automated data acquisition setup techniques are carried out to analyze the design with the location of the potential suspects in order to identify a set of registers that can provide more information about the actual bug. The newly acquired data feeds back to the proposed data analysis procedure which iterates the three steps in Figure 5.3 in the next debug session to aid in further root cause analysis.

The details of timeframe diagnosis and the techniques to aid in identifying signals to be traced in the data acquisition stage are given in the following subsections.
5.4.1 Timeframe Diagnosis

In silicon debug, the depth of the trace buffer limits the number of samples that are acquired in one debug experiment. Once the buffer is full, the older data is overwritten by the new samples. Hence, if the cycle in which the error is exercised can be estimated, the buffer can be utilized more effectively. This unique constraint motivates timeframe diagnosis.

A timeframe diagnosis pass narrows down the critical interval. This result can help to set up the next debug experiment, such that data acquisition starts at the right cycle(s) (i.e., the one(s) as close to the critical cycle as possible). Note, the test still runs from the beginning of the test vector sequence. The trace buffer is programmed to begin the capture at a later cycle.

In the following description, $INPUT(M)$ ($OUTPUT(M)$) denotes the input (output) nets of module $M$.

**Definition 5.3** Consider an ILA representation of a sequential design. A **timeframe module** $TM$ for a single module $M$ over a set of cycles $\{T_n \cdots T_{n+k}\}$ is a conceptual entity that contains the instances $M^{T_n} \cdots M^{T_{n+k}}$ of module $M$ over this set of cycles such that $INPUT(TM) = \bigcup_{t=T_n}^{T_{n+k}} INPUT(M^t)$ and $OUTPUT(TM) = \bigcup_{t=T_n}^{T_{n+k}} OUTPUT(M^t)$

Pseudo-code to identify the critical interval is described in Algorithm 5.1. Recall, in the basic SAT diagnosis process described in Section 2.4.1 the gates in each timeframe that represent the same gate in the original sequential circuit are considered as one suspect (i.e., the multiplexers associated with them share the same select line). Intuitively, one may think that it
is a timeframe module for the gate defined over all cycles. Here, instead of diagnosing suspects in one timeframe module, timeframe diagnosis examines suspects in timeframe modules that are sets-of-cycles.

Timeframe diagnosis divides the trace into several intervals of width $k$ and constructs a timeframe module for each interval. Recall, hierarchical diagnosis returns a list of suspect modules. Hence, the timeframe module considers the suspect module in each cycle of the interval as a single suspect. This is shown in lines 8–11. Consequently, timeframe diagnosis selects suspects from this new list of suspects.

The formulation of the diagnosis problem is similar to the one described in Section 2.3.1. In basic SAT-based diagnosis, one multiplexer, $m_i$, is inserted for each suspect candidate, $\eta_i$. Candidate $\eta_i$ is the suspect of the failure if the select line of the multiplexer is set. This indicates that the SAT solver can find a logic value for $\eta_i$ such that the behavior of the design matches with the expected response. In timeframe diagnosis, multiplexers are inserted for each output of the timeframe module. Furthermore, multiplexers for the same timeframe module share the same select line. Figure 5.6 depicts the design after the multiplexer insertion. Two timeframe modules, $TM_1$ and $TM_2$, are created for module $A$. The timeframe modules are defined over two cycles. Taking $TM_1$ as an example, the four multiplexers, $\{m_1, m_2, m_3, m_4\}$, have the same select line, $sel_1$; the structure is the same for the four multiplexers in $TM_2$. Hence, if the SAT solver can justify any of the suspect lines within the timeframe to make the design match with the expected behavior, the whole timeframe is selected by the SAT solver.
Figure 5.7: Timeframe diagnosis with multiple error excitation

**Lemma 5.1** Timeframe diagnosis is guaranteed to select the timeframe module with the defined interval containing the critical cycle.

**Proof:** First, since timeframe diagnosis divides the trace into consecutive intervals of cycles, the cycle when the actual error is triggered must be in one of the intervals. This implies that one of the timeframe modules must contain the critical cycle. Assume, toward contradiction, that the timeframe module that contains the critical cycle ($TM_c$) is not selected by the timeframe diagnosis. This means that the SAT solver cannot readjust the value at $OUTPUT(TM_c)$ to make the design comply with the expected output response for the given input vector. However, because the error is excited during the time interval defined by $TM_c$, the output of $TM_c$ must contain erroneous values and, correspondingly, there must be correct values. It follows that the SAT solver can assign the correct values at $OUTPUT(TM_c)$ to eliminate the error effect and make the instance satisfied. Hence, by construction, $TM_c$ must be one of the solutions returned by timeframe diagnosis. ■

To be more accurate, in addition to the timeframe module containing the critical cycle, the solution also includes timeframe modules defined over the interval between the critical cycle and the cycle in which the erroneous effects are observed. The resulting critical interval is the union of timeframe intervals defined by timeframe modules in the solution. The following example demonstrates the behavior of timeframe diagnosis.
Example 5.1 Consider a test vector interval between cycles $T_n$ and $T_{n+6}$, as shown in Figure 5.6. From hierarchical diagnosis, it is known that modules $A$ and $B$, shown here in grey boxes, are suspects. To improve the estimate for the time interval where the error is excited, timeframe modules that consider two cycles at a time (i.e., $k = 2$) are created. These timeframe modules are shown in dotted rectangles (e.g., $TM_{A1}$ consists of $\{A^{T_n}, A^{T_{n+1}}\}$). Assume that the error is excited in module $A$ at cycle $T_{n+3}$ (the grey box marked with an $\times$), timeframe diagnosis returns solutions consisting of $TM_{A2}$ and $TM_{B3}$. Hence, timeframe diagnosis can deduce that the critical interval is $(T_{n+2}, T_{n+5})$ as defined by $TM_{A2}$ and $TM_{B3}$.

Since the algorithm guarantees that one of the selected timeframe modules is the critical interval, the subsequent analysis can focus on the trace within the beginning and end cycles defined by the timeframe module solutions. In Example 5.1 because $TM_{A2}$ and $TM_{B3}$ are selected, cycles between $T_{n+2}$ and $T_{n+5}$ are analyzed in the next debug session. The value of $k$ defines a trade-off between performance and resolution. The more timeframe modules one has to examine, the more candidates that need to be considered at every iteration of the algorithm. In early debugging sessions, a larger value for $k$ may be more preferable for some coarse-grain analysis. Since failing test vectors can contain many cycles, short timeframe modules will introduce a lot of candidates that take more time to screen. On the other hand, having excessively long timeframe modules intervals may not always be a good practice at later stages.

Figure 5.7 shows a situation where the bug is excited multiple times by the test vector sequence. In the example, the bug is excited at cycle $T_n$ and $T_{n+5}$ as indicated by the $\times$’s. In this case, timeframe diagnosis will select both $TM_1$ and $TM_3$ as a solution (i.e., the cardinality of the solution is 2), since it needs to adjust the output of both modules in order to match the expected output constraint. Therefore, the critical interval for this example is $(T_n, T_{n+5})$. However, this situation does not occur in silicon debug often. The reason why functional errors escaped observation during the pre-silicon verification is because they are hard to detect. Usually, it requires a very specific or long test vector sequence to excite the error. Hence, it is seldom that such errors get excited multiple times in the system before they are captured.
5.4.2 Data Acquisition Setup

Due to the insufficient observability of internal signals, selecting which set of signals to observe is a key step in the silicon debug process. Trace buffers provide the engineer great flexibility in the choice of traced signals. However, the buffers can only trace a limited subset of signals. To make the most efficient use of this hardware, the engineer uses two major criteria for selecting signals to be traced by trace buffers:

1. Signals that are related to the error source or provide valuable information to aid in pruning suspects,

2. Signals that comply with the hardware constraints. As discussed in Section 5.2.1 in most real-world designs, only a small set of hard-wired signals can be traced during the execution.

To identify registers that may contain useful information to aid debugging, two approaches are discussed in the next subsections. The first approach is X-simulation which is an existing technique used in logic diagnosis to identify error propagation paths. The second approach is utilizing UNSAT cores that are derived from a proof of an unsatisfiable SAT instance.

5.4.2.1 X-Simulation

A simple approach to identifying registers that may be related to bugs in the design is to use X-simulation [31]. This is a simulation technique in logic diagnosis that uses a logic unknown (represented by X) to model the effects of complex faults. This technique simulates a logic unknown value on suspect nodes to capture all possible paths for error propagation. Table 5.1 shows the operation of primitive gates (AND, OR, NOT) with the presence of the logic X, where the first row and the first column represent input values to the primitive gate.

X-simulation performs cycle-by-cycle simulation. The complete procedure is demonstrated with an example shown in Figure 5.8. Assume module D is the suspect module with two outputs and the possible error propagation paths from D in cycle $T_3$ are of interest. X-simulation first performs regular (i.e., two-value) simulation from $T_1$ to $T_2$. At $T_3$, the algorithm places logic
unknown (X) at the two outputs of D and simulates the circuit for one cycle with the presence of X. Depending on the logic value of the other nodes in the design, X may or may not propagate to the registers. If X reaches the registers, there exists a path where the error effect can propagate from the suspect to the registers. In the example, an X can reach register S_3 through G_2. Hence, S_2 is the candidate for tracing.

Given a list of suspect modules \( D = \{D_1, D_2, \ldots\} \) and the critical interval \((T_i, T_j)\), X-simulation is executed for each \( D_k \in D \) and in each cycle \( T_t \), where \( i \leq t \leq j \). The collection of registers returned from each X-simulation run is the final list of register-to-be-traced.

The drawback of X-simulation is that it is a pessimistic process; that is, it may return too many registers to make the information useful. To improve the resolution and accuracy, another selection algorithm that utilizes the proof trace generated by SAT solvers is presented.

### 5.4.2.2 UNSAT Core-Based Tracing

As discussed in Section 2.1.2, an UNSAT core of an unsatisfiable SAT problem is a subset of clauses that is also unsatisfiable. Given an erroneous circuit, \( C_e \), the input vector sequence, \( v \),

---

**Table 5.1: Operation of primitives with X**

<table>
<thead>
<tr>
<th>(a) AND</th>
<th>(b) OR</th>
<th>(c) NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \land )</td>
<td>( \lor )</td>
<td>( \sim )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \land )</th>
<th>( \lor )</th>
<th>( \sim )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Figure 5.8: X-simulation**
and the correct output response, $y$, the CNF formula of the ILA representation of the circuit,
$\bigcup_{i=1}^{k}(C^i_e \cdot v^i \cdot y^i)$, where $k$ is the length of the sequence, is unsatisfiable due to the contradiction between the erroneous output response and the correct output response. Intuitively, the contradiction can occur at any signal along the paths from the actual fault location to the output where discrepancies are observed. Therefore, signals associated with clauses in the UNSAT cores must be one of the following:

- Nodes that excite the error
- Nodes along the error propagation paths
- Side inputs to the error propagation paths

Clearly, these signals can be potential locations for tracing and provide information about the erroneous behavior of the design.

Example 5.2 Consider the circuit shown in Figure 5.9(a). Assume the error is at node $i$, where the correct implementation is $i=\text{AND}(a, b)$. The test vector sequence, the correct response and the erroneous response are shown in Figure 5.9(b). Since the circuit is erroneous, the CNF formula, $\Phi = \bigcup_{i=1}^{k}(C^i_e \cdot v^i \cdot y^i)$, is unsatisfiable. For the purposes of demonstration, the CNF formula of the first cycle is shown as follows:

$$\Phi^1 = (a^1 + i^1) \cdot (b^1 + i^1) \cdot (a^1 + b^1 + i^1) \cdot (b^1 + j^1) \cdot (d^1 + j^1) \cdot (b^1 + d^1 + y^1) \cdot (e^1 + k^1) \cdot (j^1 + k^1) \cdot (e^1 + j^1 + k^1) \cdot (c^1 + g^1) \cdot (e^1 + g^1) \cdot (c^1 + e^1 + g^1) \cdot (c^1 + h^1) \cdot (f^1 + h^1) \cdot (c^1 + f^1 + h^1) \cdot (a^1) \cdot (b^1) \cdot (c^1) \cdot (d^1) \cdot (e^1) \cdot (f^1) \cdot (g^1) \cdot (h^1)$$

where Equation 5.1a–5.1e represent $C^1_e$, Equation 5.1a is the constraint on the primary inputs and the initial states, and Equation 5.1e is the primary output constraints. The CNF formula for the remaining three cycles can be derived in a similar manner.

Given the SAT instance to the SAT solver (e.g., MiniSAT [35]), an UNSAT core (Equation 5.2) of the instance can be extracted from the proof trace provided by the solver.
Figure 5.9: Example of an erroneous circuit with the correct implementation of gate \( i = \text{AND}(a, b) \)

\[
\{(\overline{b^2} + i^2 \cdot \overline{i^2} + d^3) \cdot (\overline{d^3} + j^3) \cdot (c^4 + e^4 + g^4) \cdot (j^3 + e^4) \cdot (b^2) \cdot (c^4) \cdot (g^4)\} \tag{5.2}
\]

By examining the UNSAT core, variables that represent registers can be extracted: \( d^3 \) (from the clause \( \overline{i^2} + d^3 \)) and \( e^4 \) (from the clause \( j^3 + e^4 \)). Therefore, signals that should be traced are \( d \) at cycle 3 and \( e \) at cycle 4.

The overall algorithm is shown in Algorithm 5.2 The goal is to identify as many UNSAT cores as possible and extract registers from each UNSAT core. Since each UNSAT core is one potential error propagation path, registers involved with these UNSAT cores are potentially on the error propagation paths. To obtain multiple UNSAT cores, the algorithm iteratively eliminates UNSAT cores until the problem is satisfied.
Algorithm 5.2 UNSAT core-based register selection

1: $C_e :=$ The erroneous design
2: $V :=$ Input vectors
3: $Y :=$ Output vectors
4: $\Phi := C_e \cdot V \cdot Y$

5: **procedure** IDENTIFYTRACED SIGNALS($\Phi$)

6: $U_{init} :=$ Solve $\Phi$ and extract the UNSAT core
7: $U \leftarrow U_{init}$
8: **while** $\Phi$ is unsatisfiable **do**
9: \hspace{1em} relax on clauses \{$c | c \in U_{init}$ and $c$ is an input vector clause$\}$
10: \hspace{1em} $U_{new} \leftarrow$ solve $\Phi$ and extract the UNSAT core
11: \hspace{1em} $U \leftarrow U \cup U_{new}$
12: **end while**
13: **while** $\Phi$ is unsatisfiable **do**
14: \hspace{1em} relax on clauses \{$c | c \in U_{init}$ and $c$ is an output response clause$\}$
15: \hspace{1em} $U_{new} \leftarrow$ solve $\Phi$ and extract the UNSAT core
16: \hspace{1em} $U \leftarrow U \cup U_{new}$
17: **end while**
18: $R \leftarrow$ obtain registers of which corresponding variables appearing in $U$
19: **return** $R$
20: **end procedure**
The procedure of the algorithm is as follows. It starts by obtaining the initial UNSAT core \( \mathcal{U}_{init} \) in line 6. Then, the algorithm tries to obtain more UNSAT cores through relaxation, as summarized in Section 2.1.2. First, it relaxes clauses in \( \mathcal{U}_{init} \) that represent input vectors (line 9) until the problem is SAT. Next, it repeats for clauses in \( \mathcal{U}_{init} \) that represent output responses (line 14). Since each UNSAT core can represent different error propagation paths, different signals can be included. To ensure that all paths are considered, the union of all UNSAT cores is taken, as shown in line 11 and line 16 in the algorithm. Finally, if the corresponding variables of registers appear in any UNSAT cores, these registers are the potential locations for tracing.

**Example 5.3** Continue from Example 5.2. The algorithm first relaxes the input constraints that appears in the UNSAT core. These input constraints are \( b^2 \) and \( e^4 \). Let \( r_1 \) and \( r_2 \) be the relaxation variables. The two input constraint clauses are replaced with \( b^2 + r_1 \) and \( e^4 + r_2 \), respectively. Also, additional clauses, \( (r_1 + r_2) \cdot (\overline{r_1} + \overline{r_2}) \), are added to the CNF formula to ensure only one clause is relaxed (i.e., only one of \( \{r_1, r_2\} \) can be 1). One can verify that the new formula is SAT and no additional UNSAT core can be obtained.

Next, the algorithm relaxes the output constraint, \( g^4 \). Let \( r_3 \) be the relaxation variable. The relaxed clause of \( g^4 \) is \( g^4 + r_3 \) and an additional clause, \( (r_3) \), is added to the initial formula, \( \Phi \). This new formula, \( \Phi' \), is still UNSAT and a new UNSAT core can be obtained from the proof trace, as shown in Equation 5.3.

\[
\{(a^1 + i^1) \cdot (d^3 + j^3) \cdot (d^2 + j^2) \cdot (j^2 + e^3) \cdot (b^2 + i^2) \cdot (i^2 + d^3) \cdot (d^3 + j^3) \cdot (e^3 + j^3 + k^3) \cdot (c^4 + h^4 + f^4) \cdot (k^3 + f^4) \cdot (a^1) \cdot (b^2) \cdot (c^4) \cdot (h^4)\} \quad (5.3)
\]

In the new UNSAT core, variables that represent registers are \( \{d^2, d^3, e^3, f^4\} \). Hence, the new list of registers-to-be-traced contains \( d \) at cycle 2 and 3, \( e \) at cycle 3 and 4, and \( f \) at cycle 4.

The algorithm continues to obtain more UNSAT cores by relaxing the output constraint clause, \( h^4 \), in \( \Phi' \). The clause is replaced with \( h^4 + r_4 \) where \( r_4 \) is the relaxation variable, and an additional clause, \( (r_4) \), is added to \( \Phi' \). This time, the new formula is SAT; therefore, the UNSAT core extraction is complete.
5.4.3 Test Vector Trace Reduction

In silicon debug, the length of the test vector trace affects how many iterations of chip execution need to be carried out in order to collect the trace of interest. One way to reduce the length is by comparing scan dumps and the golden model simulation results, as described in [104]. If these values match, it can be assumed that the error is excited in a later cycle. Therefore, this provides a conservative estimate for the new initial cycle for the test vector trace. However, this approach requires time-consuming scan dumps and is limited, when only a subset of the registers have their reference point in the golden model. Since the golden model is usually a high-level behavior model of the design, not every signal in the implementation can be mapped to the golden model. Hence, the reference values of some signals may not be available. If the error effect propagates through the state elements with no golden model reference, the discrepancy will not be detected early.

In order to use the aforementioned method to cut the test trace, yet ensure that diagnosis fails when the error is excited at a cycle before the new initial cycle, an additional coarse-grain module, called initial module, is introduced in the suspect list for hierarchical diagnosis. This is a conceptual module that considers all the state elements of the initial cycle as one candidate suspect. If the critical cycle is before the new truncated test vector trace, the initial module can be selected by the diagnosis algorithm to indicate that the error effects originate at a cycle before the initial cycle. In this case, the complete set of debug sessions is repeated with a new initial state at an earlier cycle.
Example 5.4 The design in Figure 5.11 contains three state elements, \{S_1, S_2, S_3\}. Assume that only \(S_1\) and \(S_2\) have a golden model reference, and that the error is excited at cycle \(T_{n-2}\) and propagated along the path, as shown in the figure. In this case, cycle \(T_n\) can be considered as the new initial starting cycle of the test vector trace during diagnosis, since \(S_1^n\) and \(S_2^n\) contain no discrepancy. That is, only the portion of the trace after \(T_n\) is used. This can result in an incorrect diagnosis result. Hence, by introducing an initial module (the dotted rectangle) that contains \{\(S_1, S_2, S_3\)\} at cycle \(T_n\), diagnosis can capture the error effects by returning the initial module as the solution. Then, diagnosis has to be restarted with a new initial cycle estimate before \(T_n\).

### 5.5 Alternative Signal Searching

The algorithm \textsc{IdentifyTracedSignals} from Section 5.4.2 selects a list of registers that may contain useful information about the behavior of the faulty chip. However, as mentioned in Section 5.2.1 only a small set of registers can be traced with the trace buffer. Those traceable registers are pre-selected during the circuit design stage. Hence, they may not have direct access to registers selected by \textsc{IdentifyTracedSignals}. Instead, an indirect approach to obtaining the value of a non-traceable register through implications from traceable registers is presented.

Consider a circuit modelled in the ILA representation shown in Figure 5.11. Let \(s^k_g\) denote the untraceable register \(s_g\) at cycle \(T_k\) of which the values are desired. Given a set of traceable registers, \(S_t\), referred to as candidate registers, the goal is to find a subset of traceable registers, \(\{S'_t \subseteq S_t\}\), such that the values of the registers in \(S'_t\) can imply the value of \(s^k_g\). Therefore,
Table 5.2: Three-value logic encoding

<table>
<thead>
<tr>
<th>Logic value</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>X</td>
<td>00</td>
</tr>
</tbody>
</table>

instead of tracing $s_g$, registers in $S'_t$ are traced. Then, the value of $s_g^k$ is restored with the values of the registers in $S'_t$. The restoration can be due to forward implications, backward justifications or both.

A SAT instance is formulated to identify these implications. The instance is satisfied if the SAT solver can assign values to a subset of candidate registers that, together with the input and output trace, imply the value of the target register. Consequently, the alternative for the target register consists of those selected candidate registers. The details of the formulation are given in the following subsections.

5.5.1 Problem Formulation

The basic formulation of the searching problem is described in this section. Let the target register be $s_g^k$. The formulation consists of two components. The first component models the circuit between $\{T_{k-w} \cdots T_{k+w}\}$. Variable $w$ is user-defined and referred to as window size. This interval constrains the search space where the SAT solver can search for implications to the target register. The second component of the formula limits the number of candidate registers used for generating implications.

In this formulation, each line of the design can have three types of logic values, $\{0, 1, X\}$, where $X$ represents unknown. This is necessary to indicate whether the value of the target register is implied. Hence, a three-value satisfiability formulation for each gate is required. It means that each node in the design is associated with two Boolean variables. Table 5.2 shows the encoding of the three logic values and Table 5.3 lists the three-value SAT formula of each gate type. For clarity, only the formulation of two-input gates is shown.
Table 5.3: Three-value SAT formula for simple gates

<table>
<thead>
<tr>
<th>Gate function</th>
<th>SAT formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p = \text{AND}(u_1, u_2) )</td>
<td>((u_{1a} + p_a) \cdot (u_{2a} + p_a) \cdot (\overline{u_{1a}} + u_{2a} + p_a)) ((u_{1b} + p_b) \cdot (u_{2b} + p_b) \cdot (u_{1b} + u_{2b} + p_b))</td>
</tr>
<tr>
<td>( p = \text{NAND}(u_1, u_2) )</td>
<td>((u_{1a} + p_a) \cdot (u_{2a} + p_a) \cdot (\overline{u_{1a}} + \overline{u_{2a}} + p_a)) ((u_{1b} + p_b) \cdot (u_{2b} + p_b) \cdot (u_{1b} + u_{2b} + p_b))</td>
</tr>
<tr>
<td>( p = \text{OR}(u_1, u_2) )</td>
<td>((\overline{u_{1a}} + p_a) \cdot (\overline{u_{2a}} + p_a) \cdot (u_{1a} + u_{2a} + \overline{p_a})) ((u_{1b} + p_b) \cdot (u_{2b} + p_b) \cdot (\overline{u_{1b}} + \overline{u_{2b}} + p_b))</td>
</tr>
<tr>
<td>( p = \text{NOR}(u_1, u_2) )</td>
<td>((\overline{u_{1a}} + p_a) \cdot (\overline{u_{2a}} + p_a) \cdot (u_{1a} + \overline{u_{2a}} + p_a)) ((u_{1b} + p_b) \cdot (u_{2b} + p_b) \cdot (\overline{u_{1b}} + u_{2b} + p_b))</td>
</tr>
<tr>
<td>( p = \text{BUFFER}(u) )</td>
<td>((u_a + \overline{p_a}) \cdot (\overline{u_a} + p_a) \cdot (u_b + p_b) \cdot (\overline{u_b} + \overline{p_b}))</td>
</tr>
<tr>
<td>( p = \text{NOT}(u) )</td>
<td>((u_b + \overline{p_a}) \cdot (\overline{u_b} + p_a) \cdot (u_a + p_b) \cdot (\overline{u_a} + \overline{p_b}))</td>
</tr>
</tbody>
</table>

Candidate registers are traceable registers within the interval, \( \{T_{k-w} \cdots T_{k+w}\} \), of which the value is not known yet. The value of some traceable registers is known because they are traced in previous debug sessions, or their values can be restored from the existing data (e.g., input vectors, observed output responses and known register values). To determine traceable registers whose value is still unknown, simple forward- and backward- logic value propagation within the cycle interval is carried out. This is done by modelling the unrolled circuit between \( \{T_{k-w}, T_{k+w}\} \) in three-value SAT. The SAT instance is constrained with the known values. The instance must be satisfied, since the erroneous design and the observed responses are used. Only traceable registers with an unknown value are considered as candidate registers.

In order to indicate whether a candidate register is selected for generating an implication, new variables, called select variables and denoted as \( L = \{l_1, l_2, \cdots\} \), are added for every candidate register at each cycle. When a select variable is assigned with logic 1, it indicates that the corresponding candidate register is used to produce the implication. The intuition behind the formulation is to allow the SAT solver to assign logic 0 or logic 1 to some of the candidate registers such that the target register is either logic 0 or logic 1 as well.

If the formula is satisfied, each solution to the problem is one possible implication for the
target register under the given input vector. Candidate registers wherein the select variable, \( l \), is active are the necessary registers to generate the implication. Because traceable registers in each cycle have one unique select variable, the algorithm identifies not only the registers, but also cycles where those registers are located in order to generate the implication.

In detail, the SAT instance can be expressed as follows:

\[
\Phi = \prod_{j=k-w}^{k+w} \Phi_c^j(\mathcal{L}^j, \mathcal{V}^j, \mathcal{Y}^j_{\text{obs}}, S^j_{\text{known}}) \cdot E_N(\bigcup_{j=k-w}^{k+w} \mathcal{L}^j) \quad (5.4)
\]

The first component, \( \prod_{j=k-w}^{k+w} \Phi_c^j(\mathcal{L}^j, \mathcal{V}^j, \mathcal{Y}^j_{\text{obs}}, S^j_{\text{known}}) \), models the design from cycle \( T_{k-w} \) to \( T_{k+w} \). Each \( \Phi_c^j \) represents a copy of the erroneous design at cycle \( j \) with input vector \( \mathcal{V}^j \) and observed response \( \mathcal{Y}^j_{\text{obs}} \), enforced at the primary inputs and the primary outputs, respectively. Previously traced register values (\( S^j_{\text{known}} \)) are also used to constrain the problem, since they may be helpful in generating implications. Registers in \( T_{k-w} \) and \( T_{k+w} \) of which values are not known are constrained with \( X \). As will be explained in the next subsection, special CNF models are required for the target register and candidate registers. Although the value \( w \) is user-defined, it also depends on the size of the trace buffer. One can set \( w \) such that \( 2w + 1 = \text{buffer depth} \) to fully utilize the memory space of the trace buffer. However, larger \( w \)'s can increase the computation complexity and memory consumption, since there are more candidate registers for selection and a larger portion of trace is analyzed. The flexibility of \( w \) allows the user to adjust it according to the available resources.

The second component, \( E_N(\bigcup_{j=k-w}^{k+w} \mathcal{L}^j) \), constrains the number of selected candidate registers. It is an adder that sums up the value of select variables. The detail of the construction can be found in [58]. To find the minimum number of candidate registers required for implications, the output of the adder is constrained to allow one active select variable, and the value is incremented until a solution is found or the total number of the select variables is reached.

The details of models for target registers and candidate registers are described in the following subsections.
5.5.2 Register Modelling

Target registers and candidate registers need to be encoded specially in the CNF formula in order to solve the searching problem described above. In this section, models applied to these two types of registers are discussed.

**Target Register:** The goal of the target register $s^k_g$ is to have a non-$X$ value (i.e., logic 0 or logic 1). The implication can come from two directions: forward propagation from assignments in the earlier timeframe, or the backward justification from assignments in a later timeframe. To allow the SAT solver to consider implications from both directions, the target register is modelled, as shown in Figure 5.12. An extra signal, $s'$, is introduced to disconnect $s^k_g$ from its fanin. If $s'$ or $s^k_g$ have a logic 0 or a logic 1 value, there exists an implication. This is stated as $cond_1$ (line 1) in Figure 5.12(b). Condition $cond_2$ enforces that the implication only needs to be satisfied from one direction. Furthermore, if there are implications from both directions, the implied values have to be the same.

**Candidate Register:** Candidate registers are traceable registers of which values are not
known yet and available for the SAT solver to select in order to generate implications. For each candidate register, two variables, \( s' \) and \( l \), are introduced, as shown in Figure 5.3(a). The select variable, \( l \), determines whether the register connects to its fanout. When \( l \) equals 0, the network remains the same (line 1-2 in Figure 5.3(b)). When \( l \) equals 1, the register is disconnected from its fanout, and the SAT-solver can assign 0 or 1 to either end of the break. This enables the possibility of identifying forward and backward implications. Similar to the model for target registers, at least one of the two variables at the disconnected ends must be either logic 0 or logic 1. If both ends have non-X values, the values must be the same.

**Example 5.5** Consider an ILA representation of a sequential circuit, as shown in Figure 5.14. Variables \( a, b, \) and \( c \) are the primary inputs; \( p \) is the primary output; \( d, e \) and \( f \) are state variables. Assume that \( d \) and \( e \) are traceable registers, and the target register is \( f^6 \). The circuit is unrolled for timeframes \( T_5 \) and \( T_6 \) (i.e., \( w = 1 \)), and the values of the primary inputs and primary outputs are shown in the bracket next to the variables. The candidate registers are \{\( d^5, e^5, d^6, e^6, e^7 \}\), which are modelled, as shown in Figure 5.13, with five additional select variables, \{\( l_1 \cdots l_6 \)\}. Register \( d^7 \) is not a candidate register because its value can be derived from the existing information (i.e., \( b^6 = 1 \) implies \( d^7 = 0 \)). Variables \( d^5, e^5, f^5, f^7, \) and \( e^7 \) are constrained with logic \( X \) since they are at the beginning and the end of the window. An unknown at these registers causes an unknown at \( f^6 \). The proposed algorithm converts this structure into SAT and asks the SAT solver to find an assignment to select variables and candidate registers.
such that $f^6$ does not have a logic $X$. In this example, one solution is \{$l_1 = 1, d^5 = 0, f^6 = 0$\}. This means that the value of $e^6$ may be restored if the value of $d^5$ is known. Another implication is \{$l_2 = 1, e^5 = 1, f^6 = 0$\}.

The algorithm only identifies the possible implications for the given condition. The value of the target registers may not be restored if the antecedent does not occur in the real-time circuit execution. For instance, if the value of $d^5$ in the previous example is actually logic 1, no conclusion can be made about the value of $f^6$. The goal of the algorithm is not to identify this exact situation in the circuit, but to provide suggestions about possible ways to obtain the value of target registers. Moreover, it is possible that no implication can be identified within the cycle interval investigated. In this case, a larger window needs to be considered.

5.5.3 Formulation Improvements

As shown in Figure 5.2(b), traceable registers are typically divided into groups. When configuring the trace buffer, one group of the traceable registers is selected and traced for several timeframes. With this observation, the number of select variables for the candidate registers can be reduced. Instead of introducing one distinct select variable for each candidate register, all registers in the same group can share a select variable. Furthermore, the same register in different timeframes can share one select variable as well. In Example 5.5, assuming $d$ and $e$ are in different groups, the number of $l$'s can be reduced to two: $d^5$ and $d^6$ share one select variable, while $e^5$, $e^6$, and $e^7$ share another select variable.

The second optimization is to find implications for a group of target registers. As mentioned in Section 5.4.2, target registers identified by the proposed method are correlated to each other. Hence, if there exists an implication for one of the target registers, the same implication may also imply the value of other target registers. By grouping several target registers together, the number of executions of the searching algorithm can be reduced. As a result, the overall runtime is reduced. However, it is a trade-off between the runtime and the precision of solutions, because more traceable registers may need to be selected when multiple registers are targeted.
5.6 Group Ranking

The algorithms described in previous sections identify registers that should be traced to provide more information about the error. Since registers are selected by groups at the end when configuring the trace buffer, a simple ranking system is described to prioritize the traceable register groups according to the results from the proposed algorithms.

- Rule 1: The group that contains the most registers returned by the algorithm IdentifyTracedSignals has the highest priority. This is because those registers are directly related to the error source. Their values may contain the most useful and direct information.

- Rule 2: When searching for alternatives for non-traceable registers, different target registers may require different traceable groups. If a group is being selected at a higher frequency than other groups, it gets a higher rank. Intuitively, this group contains registers that have a higher chance of providing implications to non-traceable registers.

- Rule 3: A higher rank is assigned to the group that needs to be traced for more timeframes. This is simply done to efficiently utilize the memory space of the trace buffer.

5.7 Extension to Physical Defects

Although the presented methodology assumes that errors in the silicon prototype are functional errors, this methodology can also deal with physical defects with minor modifications. This is possible because the methodology is a model-free approach [109]. That is, it does not make any assumptions about the behavior of the error or faults. This feature makes the methodology suitable for physical defects, since they can be complicated and difficult to model accurately.

For functional errors, the input to the methodology is an erroneous RTL model. With the help of the silicon chip, the methodology tries to correct locations in the RTL model such that it can comply with the golden reference. Hence, when constructing SAT formulae for debugging, the formulae are constrained by expected output responses. In contrast, when debugging physical defects, the RTL model is assumed to be correct (i.e., there is no functional
error). In this case, the methodology identifies the source of the error by inserting incorrect values at locations in the RTL model such that the resulting RTL model has the same erroneous behavior as the silicon prototype does. This can be done by constraining the SAT instance with the observed responses from the silicon prototype.

However, the methodology only identifies the potential lines in the RTL model where the physical error may occur. The actual cause of the failure still requires other techniques to model the failure behavior and to conduct physical inspection. In this case, mapping between a node in the RTL representation and a physical wire in the silicon is a necessity.

## 5.8 Experiments

In this section, experiments on OpenCores.org designs and ISCAS’89 benchmarks are presented. Minisat [35] is used as the underlying SAT-solver. All three processes in the proposed data analysis flow are implemented in C++ and conducted on Core 2 Duo 2.4GHz process with 4 GB of memory. All runtimes are reported in seconds. In each testcase, a single random functional error (e.g., wrong assignment, incorrect case state, etc.) is inserted into the RTL code. For designs from OpenCores.org, test vectors are extracted from the testbench provided by OpenCores.org. Test vectors for ISCAS’89 are generated randomly. In both cases, the trace length is between 100 and 300 cycles.

### 5.8.1 Hardware Setup

Since there are no existing benchmarks with trace buffers implemented, trace buffers are emulated in software in the experiments. Before the experiment begins, the erroneous design is simulated for the given test trace and the values of all traceable registers are stored in a text file.

The size of the trace buffer is assumed to be 16x128 bits. That is, it can store 16 registers for, at most, 128 cycles, or 32 registers for, at most, 64 cycles. Registers that are traceable are randomly selected. More details on these pre-selected traceable registers are discussed in each subsection. After each iteration of data analysis, the values of selected traceable registers
Table 5.4: Performance with limited hardware constraints

<table>
<thead>
<tr>
<th>Circ.</th>
<th># of reg.</th>
<th># of modules</th>
<th># of sessions</th>
<th>Total time (sec)</th>
<th>Total groups traced</th>
<th># of final susp.</th>
<th>% critical interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>divider</td>
<td>510</td>
<td>31</td>
<td>4</td>
<td>123.1</td>
<td>7</td>
<td>11</td>
<td>12%</td>
</tr>
<tr>
<td>spi</td>
<td>162</td>
<td>79</td>
<td>4</td>
<td>351.5</td>
<td>6</td>
<td>12</td>
<td>11%</td>
</tr>
<tr>
<td>wb</td>
<td>110</td>
<td>94</td>
<td>3</td>
<td>101.4</td>
<td>3</td>
<td>6</td>
<td>14%</td>
</tr>
<tr>
<td>rsdecorer</td>
<td>521</td>
<td>481</td>
<td>4</td>
<td>162.2</td>
<td>5</td>
<td>15</td>
<td>10%</td>
</tr>
</tbody>
</table>

within the determined critical interval are extracted from the stored text file. If the length of the critical interval is longer than 128 cycles, only the values in the last 128 cycles are extracted.

5.8.2 Performance of the Methodology

This set of experiments first shows the performance of the methodology. Here the algorithm is configured such that during the hierarchical diagnosis, it analyzes two levels in the hierarchy structure (n = 2) at each debug session. During the timeframe diagnosis, the trace is divided into four timeframe modules of an equal number of cycles each. X-simulation is used to determine registers that should be traced in the next debug session. It is assumed that 80% of registers in each design are traceable and that they are divided into groups of, at most, 16 registers. Since the trace buffer is 16x128 bits, in each debug session, it can store values of one group for, at most, 128 cycles or two groups for, at most, 64 cycles.

Table 5.4 outlines performance metrics for the methodology. Each experiment contains an average of five runs. The testbench used is listed in the first column. The second column contains the number of registers in the design. The third column records the number of modules at the lowest level of the hierarchy. This is also the total number of suspects one needs to examine in a brute-force manual silicon approach. The number of debug sessions and the total runtime for all sessions are shown in the fourth and fifth columns, respectively. The total number of groups that are traced is shown in the sixth column. As mentioned earlier, one or two groups of registers can be traced in each session. For example, seven groups are traced during debugging divider: one group is traced during the first session and two groups are
traced in each of the remaining three sessions. Because timeframe diagnosis often reduces the
critical intervals to more than half in the first one to two sessions, during many sessions, two
register groups can be traced in one hardware run.

The seventh column has the number of final suspects in the lowest level of the hierarchy
that engineers need to investigate. The final column has the ratio of the final critical interval
as compared to the length of the initial trace. Comparing the number of final suspects to the
number of modules shown in the third column, on average, an 85% improvement in resolution
is observed. Furthermore, the critical interval can be narrowed down to only 10% to 15% of its
initial length after the last debug session.
Table 5.5: Traceable register group information

<table>
<thead>
<tr>
<th>Circ.</th>
<th>Total reg.</th>
<th># of groups</th>
<th># of reg./group</th>
<th>% traceable</th>
</tr>
</thead>
<tbody>
<tr>
<td>spi</td>
<td>162</td>
<td>8</td>
<td>8</td>
<td>40%</td>
</tr>
<tr>
<td>hpdmc</td>
<td>453</td>
<td>16</td>
<td>8</td>
<td>28%</td>
</tr>
<tr>
<td>usb</td>
<td>2054</td>
<td>32</td>
<td>16</td>
<td>25%</td>
</tr>
<tr>
<td>s1423</td>
<td>74</td>
<td>6</td>
<td>6</td>
<td>49%</td>
</tr>
<tr>
<td>s5378</td>
<td>179</td>
<td>7</td>
<td>8</td>
<td>31%</td>
</tr>
<tr>
<td>s9234</td>
<td>211</td>
<td>8</td>
<td>8</td>
<td>30%</td>
</tr>
</tbody>
</table>

Next, the impact of two parameters of the diagnosis methodology are examined, namely the level of hierarchy that the hierarchical diagnosis examines at each session ($n$), and the timeframe module interval sizes used in the timeframe diagnosis ($k$). Figure 5.15 shows the total number of modules returned by each hierarchical diagnosis round when various numbers of hierarchy levels are examined in one debug session. In general, the numbers are increased as hierarchical diagnosis runs more rounds in one debug session. This is because there are fewer state elements provided and the diagnosis algorithm cannot distinguish some of the suspects. Figure 5.16 shows the ratio of the size of the critical interval after the last debug session compared to the original trace length when various numbers of interval are used in timeframe diagnosis. Four cases are considered: 2, 4, 8 and 16 intervals. As expected, greater reductions are achieved with finer-grain intervals. The only exception is $wb$ in the case where the interval size is 16. In this case, the error happens to be excited across two intervals, which results in a wider range. In all cases, over 50% reduction is achieved.

### 5.8.3 Performance with Hardware Constraints

This part of the experiment demonstrates the effectiveness of the UNSAT core-based register selection, as well as the searching algorithm. To emulate the real trace buffer hardware structure, a subset of registers for each design is selected randomly, or by state signal selection [100,101], as traceable by the trace buffer. These registers are divided into groups and the grouping configuration is summarized in Table 5.5. The first column lists the benchmark used
Table 5.6: Performance of debugging with proposed techniques

(a) Debug performance

<table>
<thead>
<tr>
<th>Circ.</th>
<th>No state information</th>
<th>UNSAT core-based register selection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of susp.</td>
<td># of sess’n.</td>
</tr>
<tr>
<td>spi</td>
<td>146</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>144</td>
<td>11</td>
</tr>
<tr>
<td>hpdmc</td>
<td>213</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>167</td>
<td>16</td>
</tr>
<tr>
<td>usb</td>
<td>103</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>224</td>
<td>14</td>
</tr>
<tr>
<td>s1423</td>
<td>438</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>506</td>
<td>6</td>
</tr>
<tr>
<td>s5378</td>
<td>103</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>191</td>
<td>6</td>
</tr>
<tr>
<td>s9234</td>
<td>83</td>
<td>6</td>
</tr>
<tr>
<td>Average</td>
<td>179</td>
<td>9.5</td>
</tr>
</tbody>
</table>

(b) Runtime (sec)

<table>
<thead>
<tr>
<th>Circ.</th>
<th>No state information</th>
<th>UNSAT core-based register selection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Diag.</td>
</tr>
<tr>
<td>spi</td>
<td>1990</td>
<td>828</td>
</tr>
<tr>
<td></td>
<td>179</td>
<td>101</td>
</tr>
<tr>
<td>hpdmc</td>
<td>3817</td>
<td>2323</td>
</tr>
<tr>
<td></td>
<td>2321</td>
<td>1963</td>
</tr>
<tr>
<td>usb</td>
<td>3795</td>
<td>1609</td>
</tr>
<tr>
<td></td>
<td>7091</td>
<td>4245</td>
</tr>
<tr>
<td>s1423</td>
<td>847</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>768</td>
<td>452</td>
</tr>
<tr>
<td>s5378</td>
<td>549</td>
<td>456</td>
</tr>
<tr>
<td></td>
<td>1577</td>
<td>1505</td>
</tr>
<tr>
<td>s9234</td>
<td>1042</td>
<td>1011</td>
</tr>
<tr>
<td>Average</td>
<td>1426</td>
<td>685</td>
</tr>
</tbody>
</table>

in the experiments. The second column of the table shows the total number of registers in each design. The third and fourth columns have the number of the register groups and the number of registers in each group, respectively. The fifth column shows the percentage of total registers that can be traced.

Like experiments in the previous subsection, a single random functional error is inserted into the RTL code. The algorithm is configured to perform one-level hierarchical diagnosis \((n = 1)\), and timeframe diagnosis divides the time interval into two timeframe modules \((k = 2)\). For
the searching algorithm, the window size \( w \) is set to be six timeframes and, as mentioned in Section 5.5.3, the target registers in every four timeframes are targeted together.

Table 5.6 summarizes the performance of debug analysis in two situations: debug without values of registers and debug with values of registers selected by the UNSAT core-based selection procedure. Table 5.6(a) shows the results of the debug analysis. The first column lists the benchmark used in this experiment. Each row is one individual case that contains a different bug in the design. The final row is the geometric mean of the data in each column.

The second and fourth columns show the sum of the number of modules returned at the end of each debug session under each situation, respectively. This is the total number of modules that the engineer needs to investigate. As shown in the tables, with state information, the debugging tool can effectively eliminate more false candidates in all cases. The percentage reduction on the number of suspects, comparing the second and fourth columns, is listed in the fifth column. The reduction can be as high as 78%, and an average of 33% reduction is achieved.

The number of debug sessions performed under each situation is shown in the third and the fifth column of Table 5.6(a) respectively. About one third of cases require less debug sessions to find the root cause of the error (for example, the second case of \texttt{spi}, \texttt{hpdmc} and both cases of \texttt{usb}). Finally, the number of registers traced by the trace buffer is shown in the last column. Those numbers are small compared to the total number of registers shown in Table 5.5. The benefit of the UNSAT core-based technique is shown when one considers the reductions in both the number of suspects and the number of debug sessions.

The runtime of each case is summarized in Table 5.6(b). The runtime of diagnosis under the two situations is summarized in the second and third columns, respectively. Because of the reduction of suspects and debug sessions, the runtime for diagnosis is also reduced in the case of the proposed methodology. The runtime is 52% less on average (from 1426 second down to 685 second). For the proposed methodology, the searching algorithm requires additional runtime to find registers for tracing. This search time is recorded in the fourth column of the table. As shown in the table, it can be significant in certain cases, such as \texttt{hpdmc}. This is because the algorithm has a higher failing rate on finding the recommendation for the non-traceable registers.
in those cases. The detail on the performance of the searching algorithm will be discussed later.

Overall, the total runtime of the proposed method is about 1.45 times longer than the runtime when no register data is used, as shown in the last column. However, since the number of the final suspects is reduced significantly, this additional runtime may be acceptable if there is a greater amount of time saved by manually inspecting fewer suspects.

The next experiment examines the performance of the alternative searching algorithm. Clearly, the performance of the algorithm depends on the available traceable signals. Some signals may not be able to be restored at all if the necessary registers are not traced. Hence, in addition to selecting the traceable registers randomly, state signal selection is also used. State signal selection selects registers with values that are more likely to restore other registers of which values are unknown. The results are summarized in Table 5.7. Because state signal selection only handles ISCAS benchmarks, there is no result for all OpenCores.org designs, as indicated by “–”. The second and fourth columns of Table 5.7 show the percentage of targets for which the search algorithm successfully finds alternative recommendations. The number of traceable register groups selected in order to generate application is shown in the third and fifth columns. In the case of the random selection, the algorithm is, on average, able to find an alternative for almost half of the targets. The performance of the searching algorithm in the cases where pre-selected traceable registers are chosen by state signal selection and by the random selection is similar. This is possibly because the main goal of state signal selection is
Table 5.8: Impact of state information on the diagnosis

<table>
<thead>
<tr>
<th>Case #</th>
<th>Description</th>
<th>% of suspect. reduc.</th>
<th>% of sess. reduc.</th>
<th>% of traced sig.</th>
<th>% of diag runtime reduc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X-sim with no constraint</td>
<td>82%</td>
<td>21%</td>
<td>56%</td>
<td>77%</td>
</tr>
<tr>
<td>2</td>
<td>UNSAT with no constraint</td>
<td>85%</td>
<td>26%</td>
<td>16%</td>
<td>89%</td>
</tr>
<tr>
<td>3</td>
<td>UNSAT with no search</td>
<td>27%</td>
<td>10%</td>
<td>8%</td>
<td>15%</td>
</tr>
<tr>
<td>4</td>
<td>UNSAT with search</td>
<td>31%</td>
<td>11%</td>
<td>10%</td>
<td>26%</td>
</tr>
</tbody>
</table>

Table 5.9: Impact of window sizes on the searching algorithm

<table>
<thead>
<tr>
<th>Window</th>
<th># of targets</th>
<th># of succ.</th>
<th>Succ. rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>w=2</td>
<td>5</td>
<td>2</td>
<td>40%</td>
</tr>
<tr>
<td>w=4</td>
<td>4.8</td>
<td>2.3</td>
<td>48%</td>
</tr>
<tr>
<td>w=6</td>
<td>4.1</td>
<td>2</td>
<td>49%</td>
</tr>
<tr>
<td>w=8</td>
<td>3.6</td>
<td>1.7</td>
<td>47%</td>
</tr>
</tbody>
</table>

to restore as many registers as possible throughout the whole design [100, 101]. State signal selection does not target a specific region of the design.

The next set of experiments investigates the performance of debugging when various state information is available. The experimental results are summarized in Table 5.8. All numbers are the averages of the 11 erroneous benchmarks discussed in Table 5.6. The reference case for comparison is the case wherein no state information is used (Table 5.6). The first column lists the four considered cases. The next two columns summarize the reductions of the number of suspects and the number of sessions, respectively. The fourth column is the ratio of traced registers to the total number of registers, followed by the percentage reduction of the diagnosis runtime in the fifth column.

To demonstrate the advantage of the proposed UNSAT core approach, the approach is compared with the X-simulation, as shown in cases 1 and 2. In these two cases, no hardware constraints are considered; that is, all registers are assumed to be traceable. The table shows that the UNSAT core approach outperforms the X-simulation approach in all columns, particularly with respect to the number of traced registers. This demonstrates that the UNSAT core approach can achieve better performance with less state information.
Figure 5.17: Performance of the search algorithm with three trace buffer group configurations. All have the same number of groups, but the number of registers per group is 4:2:1

For cases 3 and 4, only debugging with the UNSAT core approach is considered, as well the trace buffer hardware constraints. However, in case 3, the searching algorithm is not executed to find alternatives for non-traceable registers. Those registers are simply ignored. Comparing the results of cases 3 and 4, one can see that, with the help of the searching algorithm, the debugging performs better. For example, the reduction of suspects increases from 27% to 31%. This implies the effectiveness of the searching algorithm.

In the last set of experiments, two variations of the experiment setup are implemented to further investigate the performance of the searching algorithm, namely, the search window size
First, the algorithm is executed with four different window size values \( w = 2, 4, 6, 8 \). The performance metrics are summarized in Table 5.9. The first column lists the four window size's considered. The second column shows the average number of targets of all testcases. The third column shows the average number of targets that the searching algorithm can successfully find an alternative, and the success rate is shown in the fourth column. One can observe that as the window size becomes larger, the number of targets decreases. This is because more input and output values are applied when a larger window is used, which provides extra information to imply values to some targets for which are unknown under the smaller window of the trace. Furthermore, in general, a higher success rate is achieved as the window size increases. This is expected since there are more candidates for selection and a longer trace is used, which can restore the values of more signals.

Next, the effect of three different hardware group structures on the performance of the searching algorithm is investigated. Config_1 is the configuration in Table 5.5. Config_2 and Config_3 have the same number of traceable groups as Config_1 does, but the number of registers in each group is only a half and a quarter of the size of Config_1, respectively. For instance, Config_1 of hpdmc has 16 groups that are the size of eight registers; Config_2 has 16 groups that are the size of four registers, while Config_3 has 16 groups that are the size of two registers.

The success rate for finding an alternative of non-traceable registers is plotted in Figure 5.17(a). As expected, since there are fewer traceable registers, more non-traceable registers cannot be replaced. Hence, the success rate drops as the number of candidates become fewer. Figure 5.17(b) depicts the average number of selected traceable groups for generating implications. In general, more traceable groups are required when each group contains fewer registers (for instance, Config_1 and Config_2 of s1423).

However, cases such as Config_3 of s1423 require fewer traceable groups on average. This is because the average number is calculated for target registers that the algorithm successfully identifies as an alternative recommendation. When fewer registers are available for tracing, there is a higher chance that the algorithm will fail to find an alternative. Consequently, the average number of selected traceable groups can be smaller.
5.9 Summary

Automated software silicon debug solutions are a necessity today to ease the tasks of the test and design engineer during chip failure analysis. In this chapter, a set of software techniques for silicon debug is presented. These techniques aid in data analysis in the iterative debug sessions of the silicon debug flow. In addition to pruning the suspects of the root cause in both spacial and temporal domains, techniques that help to set up data acquisition environments are presented. The experimental results confirm the effectiveness of the approach and also demonstrate that the methodology maintains good performance under the data acquisition hardware limitation.
Chapter 6

Conclusion

As VLSI designs continue to increase in size and complexity, developing an error-free design becomes a challenging task. To ensure a systematic, robust design flow, CAD tools are required to automate various processes in the development of VLSI designs. Those tools not only make the design process more efficient, but enhance the reliability of the end product. Advances in the development of CAD tools are one of the keys to the rapid growth of the IC industry.

Despite the use of CAD tools, it is still unavoidable that errors will reside in the design due to human factors, bugs in the tools, etc. When erroneous behavior of designs is observed, it is important to identify the cause of the error within a short turnaround time and provide solutions to the error. Therefore, there is a need for automated debugging techniques that can provide accurate and efficient debugging analysis. Moreover, as designs become more complex, there is an increased demand from consumers for quality products. Hence, new methodologies that fine-tune designs for better performance and that improve designs to be more error-tolerant are needed.

This dissertation addresses the automation of three processes in the VLSI design flow that relate to improving the quality of the design: logic restructuring, logic optimization for soft error rate and data analysis for silicon debug. The dissertation’s contributions are summarized as follows.

During the VLSI design process, a synthesized design often requires modification in order to accommodate different goals. In order to preserve the engineering efforts that have already
been investigated, a dynamic logic restructuring algorithm is presented in Chapter 3. This chapter first introduces the concept of approximate SPFDs (αSPFDs), which contain a subset of information provided by the original SPFDs. A simulation-based procedure for computing αSPFDs is outlined. By approximating SPFDs, the runtime or memory issues that formal methods of computing SPFDs may encounter can be alleviated. Then, the flexibility expressed by αSPFDs is used to restructure the local network of a design to comply with new constraints. The αSPFD-based logic restructuring algorithm is proposed for both combinational and sequential designs. It restructures the local network with additional fanins to make the new design comply with the required output response. Those fanins are identified according to the αSPFDs of the network. Two searching approaches are outlined for this task: the SAT-based approach returns an optimal solution while the greedy approach returns sub-optimal solutions with shorter run-time. An extension to perform logic restructuring at multiple locations simultaneously is also discussed. Extensive experiments show that the proposed αSPFD-based logic restructuring technique is able to construct transformations at locations where other methods fail. For combinational designs, the proposed methodology can restructure five times more locations than a dictionary-based methodology; for sequential designs, the methodology can restructure up to 93% of the potential transformation locations. Furthermore, results also show that while the success rate of restructuring increases as more simulation vectors are used, only a small number of vectors is required for accurate results.

As the VLSI technology scales toward 45nm process and beyond, logic designs are becoming even more vulnerable to soft errors produced by noise and radiation effects. A logic-level optimization algorithm for soft error rate (SER) minimization using αSPFD-based rewiring is outlined in Chapter 4. It first shows that the number of conditions where transient errors on a wire can affect the design is equal to the size of the care-set of the minimum SPFD of the wire. Using αSPFDs to explore the design space, the algorithm reduces the size of the care-set by rewiring a node to produce a functionally equivalent but structurally different design. Equivalently, this restructure introduces redundancies into the design and increases the chance of logic masking when soft errors occur. Experimental results show that the SER of the design can be reduced substantially when SER reduction is the only objective in logic optimization.
The SER of half of the benchmarks is reduced by more than 15%. However, experiments also show that this reduction can come with the expense of increased delay or reduced fault coverage. In other cases where all design parameters are considered, a significant reduction in the SER can still be achieved in some cases. Experimental results show that in those cases it also results in significant reduction in one or more of other design parameters.

Silicon debug has become a necessary step in the VLSI development flow due to the fact that design errors can unavoidably escape pre-silicon verification and pass in the silicon prototype. In Chapter 5 techniques are developed to aid in identifying the root cause of failures for silicon debug. Different from most prior research work that tackles the silicon debug problem from a hardware perspective, this work suggests an automated software flow for analyzing the data collected during the silicon test. The flow includes hierarchical diagnosis to locate the potential modules that cause the erroneous behavior, timeframe diagnosis to determine the time interval where the error triggers, and a SAT-based approach to configure debug hardware in the chip for the next debug iteration. The proposed root cause analysis starts with a coarse-grain resolution. After each debug session, a finer resolution is obtained. By removing false suspect candidates in the early iteration of the debug process, the engineer can save time and avoid investigating false suspects. Furthermore, the proposed work also addresses the hardware configuration for data capturing. First, an approach that utilizes UNSAT cores to identify the nodes related to the sources of failures is outlined. Compared to other tracing techniques, the set of signals selected by the proposed approach is more concise and effective. Second, to comply with the trace buffer hardware constraints, a SAT-based searching approach is presented to find alternative signals when the signal of interest is untraceable. Then, the value of the signal of interest can be derived by implications. Experiments demonstrate that the proposed flow can effectively reduce both the number of suspects and the number of sessions required: an average of 31% suspect reduction and 12% session reduction.
6.1 Future Work

This thesis provides the basis, theory and empirical evidence confirming the effectiveness of the proposed methodologies for automating logic restructuring and the data analysis process of silicon debug. In particularly, silicon debug is an emerging issue garnering attention from both academia and industry. There are considerable improvements to be developed to improve the performance of the silicon debugging process, such as the accuracy of debugging solutions, or the capability to handle large designs and longer vector traces. Directions for future work in logic restructuring and silicon debug are summarized below.

While the proposed $\alpha$SPFD-based logic restructuring methodology achieves promising results for combinational designs, the success rate of viable solutions in sequential designs is relatively lower. This is mainly due to the uncertainty encountered during constructing $\alpha$SPFDs. Minterms that should be in the care-set of $\alpha$SPFDs may be labelled as don’t cares because the value of the new transformation under those minterms cannot be determined. Hence, an approach that can accurately determine the don’t cares in $\alpha$SPFDs is necessary to improve the quality of the restructuring. An obvious approach is to use longer test sequences such that no ambiguous state transition exists. However, in this case the algorithm may encounter a memory issue since more SPFD edges need to be considered. Further study is required to find the true set of don’t cares in $\alpha$SPFDs for sequential designs. Note that the proposed methodology only restructures the combinational portion of sequential designs. That is, the state elements are not modified. Hence, another interesting extension of the proposed work is to restructure state machines (e.g., additions or removals of sets of states for retiming). This promises a new set of possibilities in algorithmic restructuring for sequential designs.

Scalability is still a major issue for the proposed silicon data analysis. Any techniques that can reduce the length of the initial input trace or reduce the size of design under debug are beneficial to the proposed analysis techniques. Some ideas that can be investigated to handle long trace lengths are briefly discussed below:

- **Vector compression.** Although the input trace can contain thousands of cycles, the actual sequence that is required to reach the crash state from the initial state may be only a few
hundreds of cycles. Vector compression analyzes the state machine and finds a shorter path between two states in the original state sequence. The challenge of this approach is to identify the key states and ensure that they are retained in the compressed trace.

- **Vector extraction.** Instead of finding a shorter path that can lead to the same crash state, one can extract the portion of the trace that is guaranteed to contain the event that excites the error. Next, the analysis can be applied only to this smaller portion. A similar approach has been proposed, which performs a binary search based on the comparison between the scan values and the simulation results. However, this approach depends on the scan dump operation, which is time consuming, and a more efficient approach should be investigated.

- **Backward analysis.** The aforementioned two approaches may require the engineer to take in the complete initial trace before performing any trace reduction. This may not be feasible if the initial trace is too long. An alternative method is to carry out the analysis backwards from the crash state or erroneous outputs. As such, the trace is expanded only if necessary, so that the trace is kept as short as possible. However, in extreme cases, this approach may not work. For instance, if the error is excited at the beginning of the sequence and can only be observed at the end of the trace, the whole trace needs to be expanded in order to accurately debug the design.

Note that the objective of the proposed debug flow is to obtain a coarse-grain solution at early iterations of the debug process. Hence, in the early stage, one can reduce the size of the design by abstracting the implementation details of suspect modules. Then, the implementation of the module is refined in the next session if the module is determined to be a suspect of the failure.

Another interesting direction for future work is to develop an algorithm to design the trace buffer hardware structure. There have been several research works on automating the selection of registers that should be made available to be traced by the trace buffer (e.g., *state signal selection* that is used in this dissertation). The objective of these works is to maximize the number of non-traceable registers that can be restored by the value of traceable registers over
the whole design. The searching algorithm presented in this dissertation can also be adopted to perform the same task. The new objective is to find a set of registers with values that can imply the highest numbers of non-traceable registers. This selection algorithm can be complementary to the searching algorithm. With such a set of registers available, it is easier for the searching algorithm to find an implication of a non-traceable register during debugging. As a result, the performance of the algorithm can be further improved.
Bibliography


[31] V. Boppana and M. Fujita, “Modeling the unknown! towards model-independent fault 


[33] S. Cook, “The complexity of theorem proving procedures,” in Symp. on Theory of Com-
puting, 1971, pp. 151–158.

[34] J. P. Marques-Silva and K. A. Sakallah, “GRASP: a new search algorithm for satisfiabil-


guided ATPG solver and its applications for solving difficult industrial cases,” in Design 

95–98.


