On-chip Tracing for Bit-Flip Detection during Post-silicon Validation

ON-CHIP TRACING FOR BIT-FLIP DETECTION DURING POST-SILICON VALIDATION

BY

AMIN VALI, B.Sc., M.Sc.

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

© Copyright by Amin Vali, December 2017

All Rights Reserved

Master of Applied Science (2018) McMaster University
(Electrical & Computer Engineering) Hamilton, Ontario, Canada

TITLE: On-chip Tracing for Bit-Flip Detection during Post-silicon Validation

AUTHOR: Amin Vali
M.Sc., (System on-chip Design)
KTH - Royal Institute of Technology, Stockholm, Sweden

SUPERVISOR: Dr. Nicola Nicolici

NUMBER OF PAGES: xix, 144

To my supportive & lovely family

Abstract

Post-silicon validation is an important step during the implementation flow of digital integrated circuits and systems. Most of the validation strategies are based on ad-hoc solutions, such as guidelines from best practices, decided on a case-by-case basis for a specific design and/or application domain. Developing systematic approaches for post-silicon validation can mitigate the productivity bottlenecks that have emerged due to both design diversification and shrinking implementation cycles.

Ever since integrating on-chip memory blocks became affordable, embedded logic analysis has been used extensively for post-silicon validation. Deciding at design time which signals to be traceable at the post-silicon phase, has been posed as an algorithmic problem a decade ago. Most of the proposed solutions focus on how to restore as much data as possible within a software simulator in order to facilitate the analysis of functional bugs, assuming that there are no electrically-induced design errors, e.g., bit- flips. In this thesis, first it is shown that analyzing the logic inconsistencies from the post-silicon traces can aid with the detection of bit-flips and their root-cause analysis. Furthermore, when a bit-flip is detected, a list of suspect nets can be automatically generated.

Since the rate of bit-flip detection as well the size of the list of suspects depends on the debug data that was acquired, it is necessary to select the trace signals consciously.

Subsequently, new methods are presented to improve the bit-flip detectability through an algorithmic approach to selecting the on-chip trace signals. Hardware assertion checkers can also be integrated on-chip in order to detect events of interest, as defined by the user. For example, they can detect a violation of a design property that captures a relationship between internal signals that is supposed to hold indefinitely, so long as no bit-flips occur in the physical prototype. Consequently, information collected from hardware assertion checkers can also provide useful debug information during post-silicon validation. Based on this observation, the last contribution from this thesis presents a novel method to concurrently select a set of trace signals and a set of assertions to be integrated on-chip.

Acknowledgements

I use this opportunity to thank those who have affected me or my life and helped me get where I am today. First and foremost, I thank my family who have always loved and supported me at every stage of my carrier. My parents have worked tirelessly to raise me and my siblings could not have been any kinder to me.

I express my gratefulness to my PhD supervisor, Dr. Nicola Nicolici, who has always guided me throughout the years and from whom I have learned immensely in a variety of domains such as academic, professional and above all ethical work. His patience, tested on many occasions, and his humane attitude, which is the single most important factor one can look for in a supervisor, are not easy to find.

I am grateful to my supervisory committee members, Dr. Aleksander Jeremic, Dr. Shahin Sirouspour and Dr. Alexandru Patriciu, as well as my external examiner Dr. Masahiro Fujita, for their valuable input from which my thesis benefited significantly. I also thank the administrative and technical staff of the ECE department at McMaster, whose hard work ensures that the department runs its intended tasks and maintains its academic objective. Many thanks to Dr. Tim Davidson, the current chair of the ECE department who has generously spent time with me to listen, teach and support and has maintained an open and energetic environment within the department.

My time as a graduate student in the Computer Aided Design and Test (CADT) lab overlapped with many friends, colleagues and office-mates who have each helped me either with their valuable feedback in the group meetings, or kept me sane with their company through graduate school years, which many believe to be among the toughest periods of one's life. My sincere appreciation goes to Dr. Henry Ko, Dr. Adam Kinsman, Dr. Zahra Lak, Dr. Jason Thong, Dr. Pouya Taatizadeh, Dr. Xiaobing Shi, Phil Kinsman, Yasamin Fazliani, Trevor Pouge, Stefan Dumitrescu, Alex Lao, Karim Mahmoud and Pooyan Mehrvarzy. I particularly acknowledge Pouya not only for his companionship as a friend but also for his valuable input for parts of my work on hardware assertion generation. Lastly, I am indebted to Hoda Rezaee Kaviani for her continuous and loyal support, for which I am deeply grateful.

I know that I will regret it for the rest of my life if do not thank a few of my most influential educators from my pre-grad-school years. I thank Dr. Mohammadreza Jahangir, Dr. Zain Navabi, Dr. Shams Mohajerzadeh and Mr. Nasser Mohammad Khanlou for their key role in my education and shaping my character.

Notation and abbreviations

BIST	Built-in Self-Test
CAD	Computer Automated Design
CUT	Circuit Under Test
DFT	Desgin for Testability
DUV	Design Under Verification/Validation
EDA	Electronic Design Automation
FPGA	Field Programmable Logic Array
HDL	Hardware Description Language
IC	Integrated Circuit
ILP	Integer Linear Programming
IO	Input/Output
LFSR	Linear Feedback Shift Register
RTL	Register-Transfer Level
VLSI	Very Large Scale Integration

Contents

Abstract iv

Acknowledgements vi

Notation and abbreviations viii

1 Introduction 1

1.1 Design Methodology 4

1.1.1 Behavioural Design 5

1.1.2 Datapath Desgin 5

1.1.3 Logic Desgin 5

1.1.4 Physical Design 6

1.1.5 Manufacturing 6

1.2 Test Methodology 7

1.2.1 Pre-Silicon Verification 7

1.2.2 Post-Silicon Validation 10

1.2.3 Manufacturing Test 12

1.3 Thesis Organization 14

2 Background and Related Works **17**

2.1 Why Post-Silicon Validation? 18

2.2 Major Challenges 18

2.2.1 Controllability and Observability 18

2.2.2 Simulation and Golden Response 19

2.2.3 Reproducibility 19

2.3 Notable Solutions 20

2.3.1 Scan-chains 20

2.3.2 On-Chip Stimuli Generation 21

2.3.3 Error Detection 23

2.3.4 On-chip Trace Buffers 25

2.3.5 On-Chip Tracing in the Presence of Electrical Bugs 28

2.3.6 The Debug Process and the Scope of This Work 30

2.4 Overview of Contributions in this Thesis 32

2.4.1 Automatic Detection of Bit-Flips 32

2.4.2 Trace Signal Selection 34

2.4.3 Concurrent Trace and Assertion Selection 35

3 Satisfiability-Based Test Platform **37**

3.1 Background 37

3.1.1 The Scope of This Chapter 38

3.1.2 Motivational Examples 39

3.1.3 Assumptions and Nomenclature 41

3.1.4 SAT Formulation and Its Use for Post Silicon Debug 43

3.2 Methodology and Evaluation Platform 47

3.2.1 Circuit Unrolling 48
3.2.2 Translation 49
3.2.3 Merging and Running the SAT Solver 50
3.2.4 Backward Translation and Filtering 51
3.2.5 Evaluation Platform 51
3.3 Experimental Results 53
3.3.1 The Non-Uniqueness of the Core of UNSAT 57
3.4 Summary 58

4 Trace Signal Selection **60**
4.1 Background 60
4.2 Motivational Example 61
4.2.1 Restoration Ratio vs Bit-flip Detectability 61
4.2.2 Key Insight 65
4.3 New Trace Signals Selection Algorithm 67
4.3.1 Definitions 68
4.3.2 Certainty Propagation Rules 71
4.3.3 The Heuristic for Trace Signals Selection 84
4.4 Results 86
4.4.1 Experimental Setup 87
4.4.2 UNSAT Rate: Bit-flip detectability 88
4.4.3 Core of UNSAT: Size and Time Distribution 90
4.4.4 Comparison with Restoration-based Method for Trace Signals Selection 92
4.4.5 Assessing Different Algorithm Parameters 95

4.5 Summary 97

5 Joint Selection of Trace Signals and Assertion Checkers 99

5.1 Background 99

5.2 Motivation and Definitions 102

5.2.1 Example 103

5.2.2 General Objective 104

5.2.3 Key Insights 105

5.3 Co-Selection Algorithm 108

5.3.1 Main Algorithm and Cost Function 108

5.3.2 Bit-Flip Detection Probability 111

5.3.3 Integration of Assertion Benefits 114

5.4 Results 116

5.4.1 Experimental Setup 116

5.4.2 Bit-flip Detection Rate versus Hardware Cost 117

5.4.3 Different Selection Strategies 121

5.4.4 Time Distribution of the Suspects' List 121

5.4.5 Runtime 123

5.5 Summary 124

6 Conclusion and Future Work 126

List of Figures

1.1 Design Flow of Digital Circuits. Boxes show the steps and the bullet points are the corresponding output of each step. Reproduced from [1]. 4

1.2 Test Flow of Digital Integrated Circuits. 7

1.3 Shmoo Graph of Intel's ATOM processor. [*] characters represent a pass, other characters show various types of failures [2]. 15

2.1 Illustration of the serially connected scan chain. Each Scan-Flip-Flop consists of a regular flip-flop attached to a multiplexer that enforces the normal operation mode or a test mode. Reproduced from [3] . . . 20

2.2 Added on-chip hardware can improve controllability and observability. Some methods such as Scan Chains help with both, while others focus only on a specific aspect. 22

2.3 The software on the left processed the circuit and generates a set of cubes according to the verification requirements. The cubes contain fixed bits (0/1) and flexible bits(x) that can be randomly filled with either 0 or 1. The cubes are then loaded to a hardware memory. The correction logic receives a pseudo-random sequence of bits and matches it to the received cube, basically filling the x bits with 0/1. 24

2.4 Pre-silicon and post-silicon tasks for trace-based debugging. 31

3.1 Pre-silicon and post-silicon tasks for trace-based debugging. Last step is to process the collected trace and find out the root cause of the bit-flip. 39
3.2 An example circuit with consistent values on the left-hand side and an illegal combination on the right-hand side. 40
3.3 (a) Carry and Sum are both 1. (b) Unsuccessful attempt to find A and B due to logic inconsistency. (c) Equivalent SAT problem. First line describes the `AND` gate, second line the `XOR` and the third line is derived from the collected trace bits. In conjunctive normal form (CNF) all clauses must conjunctively evaluate to 1. 42
3.4 The SAT formulation consists of two sets of SAT clauses: From the circuit structure and from the failing trace. 48
3.5 All the flip-flops are unrolled for as many times as the trace depth. . 49
3.6 Flip-flops are replaced by error-injectable flip-flops. 52
3.7 Bit-flips are injected to the circuit-under-test. 52
3.8 Ratio of UNSAT problems vs the ratio of signals being traced. 54
3.9 Distribution of the candidate group in time with respect to the bit-flip injection at cycle 0. 55
3.10 Distribution of the number of flip-flops in the candidate group for two trace widths (left:128 and right: 256). 56
3.11 Histogram for the runtime of the SAT solver, in seconds, for SAT and UNSAT problems (left: SAT, right: UNSAT). 57
4.1 Pre-silicon and post-silicon tasks for trace-based debugging. Choosing which signals to trace is covered in this chapter. 62

4.2 Example circuit with trace table. Traced signals are marked with [T]. (a) Shift register with 3 flip-flops and flip-flop A is traced. (b) Trace bits recorded in the on-chip memory. (c) Same as (b), but after data restoration. (d) Trace table when bit-flip occurs in flip-flop B. (e) Flops A and C are traced. (f) Bit-flip error is detected and marked with E. . . 64
4.3 (a) Circular shift register. Forward and backward propagations are marked. (b) Error detected using multiple bits of a single trace signal. 65
4.4 (a) T1-3 partially restore circuit's state. (b) Signals in the intersection of T4-6 can be implied twice, hence a possibility for error detection. We assume a second restoration originates and propagates from a second port/path. 66
4.5 (a) Representation of Certainty (C) and Value (V) as probabilities. (b) Examples I-III showing how Y is only partially zero-restorable. . . 70
4.6 (a) Flip-flop receives certainty from D and Q. Dashed squares indicate storage that keeps the highest certainty received so far. (b) 2-input logic gate receives 3 values. (c) Gate K has a fan-out of 2. Each pair collectively restore the third member. (d) Highlighted area is certainty region of M. (e) Branches of multi-fan-out points are numbered, where the **stem** is labeled **Branch 0**, and the n output branches are labeled Branch 1 to n (refer to Algorithm 4). 73
4.7 Probe Data Structure 75

4.8 Various types of propagation: "**1**" represents the initial request which causes another propagation "**2**". (a) Trace flip-flop initiates forward and backward propagation. (b) Backward to Q-port is echoed back from D. (c) Similarly, forward to D-port is echoed forward through Q. (d,e) Similar to (b) and (c) but for a logic gate. 78
4.9 Various types of propagation in presence of branches. (a),(b) Illustration of Rule 3: A backward propagation on a branch causes forward propagation to other branches as well as backward to the stem. (c),(d) Examples of combination of Rule 3 with Rules 1 and 2. 82
4.10 Propagation stops when a complete loop has been traversed. (a) In the third step the loop is detected. CV_D is updated and propagation stops. (b) Initial request causes two propagations, backward and forward, 2b and 2f. Each circulates for one loop and stops. 84
4.11 A simplified view of a typical circuit block from `s35932` 90
4.12 Left: Rate of UNSAT problems vs. percentage of flip-flops to trace. Right: Percentage of covered flip-flops after 5 injections. 91
4.13 Distribution of the size of the suspect flip-flop list in the core of UNSAT. 91
4.14 The difference in clock cycles between the injection time (t=0) and the time-step of the suspected signals in the core of UNSAT. 92
4.15 The effect of reducing threshold for certainty comparisons for `s9234` with $\Delta = \{0.3, 0.2, 0.1 : 10^{-6}\}$. 93

4.16 Left: Rate of detected bit-flips vs. the percentage of flip-flops to trace. Right: Percentage of covered flip-flops after 10 injections. Circuit under test is `s15850` with 534 flip-flops. A 10-15% improvement is consistently observed for the evaluated circuits. 94
4.17 Various configurations for selecting signals (see Table 4.4). 97
5.1 Big picture: Pre-silicon and post-silicon tasks for collaborative trace-and-assertion-based debugging. In pre-silicon the desired signals to be traced and the assertion checkers to be synthesized must be selected. During post-silicon validation, the collected trace is extracted as well as the violations caught by the assertion engine to indicate any of the assertions that fired during the debug session. All of this debug information is then post analyzed to generate a list of suspects, useful for root-cause analysis. 100
5.2 A half-adder cell with trace buffer and a single assertion property checker. Assertion checkers get violated when both C and S are logic 1, which is an illegal state for this circuit. Trace buffer is recording a history of only two nets, A and C. 102
5.3 Trace buffer (TB) and the assertion engine (AE) collect real-time data from circuit under test (CUT), sharing wires to minimize routing costs. TB is assumed to trace only flip-flops from CUT and wires fed to AE. 104

5.4 Left: Figurative diagram of detecting bit-flips. All the possible bit-flips (every flip-flop, both $\downarrow_0^1$and $\uparrow_0^1$) might be detected by either violating an assertion checker, or with the help of several traced signals, or both, or in the worst case scenario stay undetected. Key insight in this work is to reduce the overlap and spend the budget to cover more bit-flips. Right: Same idea for circuit in Fig. 5.2 106

5.5 (A) Bit-flip coverage after 10 injections for `s5378` versus estimated hardware cost. Lower data points connected with a dashed line have no assertion checkers to boost the detection. Each data point is a 10-bit-flip injection experiment. (B) Average count of suspect flip-flops for experiment in A. 118

5.6 Similar experiments as in Figure 5.5 for a different circuit: `s38417`. Bit-flip coverage after 10 injections and average number of suspect flip-flops versus the estimated hardware cost. 119

5.7 Bit-flip detection versus wire-count of the selected set of assertions, for various selection strategies; such as selecting all assertions first and then all the traces. Dynamic selection (co-selection algorithm) performs better. 120

5.8 The order of decisions to select Assertions, wires from assertions, and flip-flops for circuit `s5378`. Wire budget for assertions is 64 and wire budget for trace selection is 48. 122

5.9 Time distribution of the flip-flops in the list of suspects for `s38417` assuming that the injection time is at time T=0. Green bars represent the distribution for when only trace information is used. Blue line illustrates the same distribution for when both violated assertions and trace signals are used for post analysis. Left: Trace width of 64 bits. Right: Trace width of 256 bits. 123

Chapter 1

Introduction

According to the Merriam-Webster's dictionary the term "computer" was first used in 1613 for humans whose job was to perform mathematical calculations[4, 5]. However in the past two centuries the word has changed its meaning to machines that have revolutionized the world by their ability to run cumbersome tasks at a much faster pace than what is imaginable for a human brain.

While it is not easy to pin-point which computer in the history is actually the first one, we can still enumerate a few of the most remarkable attempts to build such a machine. The first mechanical computer was proposed by Charles Babbage in 1822 and due to its complexity and overwhelming cost, was not built until 1910 by his son, Henry Babbage. His machine would work on a decimal basis and was programmable using punched cards [6].

The first electro-mechanical computer is believed to have been built by Konrad Zuse in Germany by 1938 [6]. This machine, called "Z1", is a binary programmable computer and a first of its kind in the history of modern computers. It took a few more years before the first pure-electrical computer came into existence. Colossus, an

electronic programmable digital computer, was used by British codebreakers in 1945. It contained no memory and used vacuum tubes to perform operations. The Electronic Numerical Integrator and Computer (ENIAC), was a fully electronic vacuum tube-driven programmable computer with a memory unit. Its 50 tons of components consumed 150kW of power and was expanded over 1800 square feet [6]. It was not however until 1960's when transistor-based computers were built which could resolve the major bottleneck until then: integration of components.

Following the invention of first transistor in 1949 and the first integrated circuit (Fairchild Semiconductor, 1960), new horizons opened up for manufacturing compact low power computers suitable for day-to-day personal use. In the 1960's, digital computers with a variety of sizes and computing powers were built using discrete components. In early 1970's Intel's 4004 general purpose microprocessor was introduced. In an attempt to build a chip that was supposed to solve a very specific problem, engineers in Intel developed a chip that was capable of running a program stored in an external memory, hence could be labeled as a general purpose processor [7]. With a clock speed of less than a megahertz and employing a $10\mu m$ process, this was a major turning point in the history of computing. The layout of the 2300 transistors in this integrated circuit (IC) was drawn manually by cutting sheets of Rubylith into thin strips [7].

It was around the same time when Moore's law [8] started to gain reputation. Gordon Moore, co-founder of Fairchild Semiconductor and Intel predicted that the number of components integrated in a single chip will double every year. A decade after Moore's initial prediction, it was suggested that doubling of component count occurs closer to every two years. Although Moore's law is more of an observation

rather any physical law, it has been called a "law" since the observed trend has been valid for several decades.

Intel's 4004 microprocessor contains 2300 transistors while today's high end processors incorporate billions of transistors on a single die. For example GV100 Volta GPU from Nvidia consists of 21 billion transistors [9]. A back of the envelope comparison between this two processors shows a 2^{23} fold increase of integration over a span of 45 years. The group of Intel engineers managed to overcome the design complexity of 2300 transistors manually. They had to design the digital circuit, draw its physical layout, manufacture, test and debug it to validate proper functionality.

While those manual tasks were successfully accomplished in less than two years [10], it is impossible to scale them for today's demands without extensive use of modern automated tools. Nowadays, we rely on powerful computers and many families of computer-automated design (CAD) tools to design the next generation of technology. Graphical tools are used for laying out the circuit while most of the steps have been automated to minimize human involvement. Other electronic design automation tools (EDA), such as various types of simulators or automatic test pattern generators (ATPG), also play an essential role in today's Very Large Scale Integration (VLSI) industry. Historically, as the complexity of electronic devices rises, many fields of research have opened up to provide systematic solutions.

Below we will summarize the main steps needed to manufacture electronic integrated circuits.

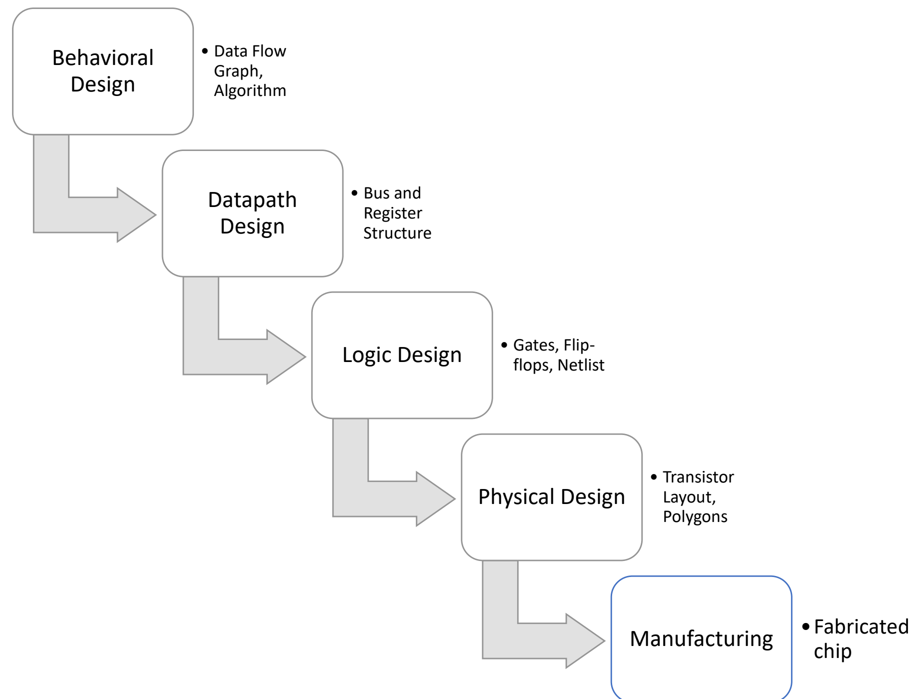


Figure 1.1: Design Flow of Digital Circuits. Boxes show the steps and the bullet points are the corresponding output of each step. Reproduced from [1].

1.1 Design Methodology

Before a digital integrated circuit is ready to be used by the end consumer, there are several steps involved in the process, as introduced briefly here. A digital IC is first designed and then its functionality is tested thoroughly for compliance with the original requirements.

According to [1] the digital design procedure can be broken into several steps once a design idea is in place and before the functional circuit is ready. Figure 1.1 shows these steps and the expected output of each step.

1.1.1 Behavioural Design

It is assumed that a set of design requirements and specifications of the desired system are provided to the designer. For example one might need a video decoder with certain input and output format and performance requirements. In this step the designer creates a high-level solution such as an algorithm, pseudocode or software model. High-level description languages such as SystemC [11] may be used to abstract away many of the implementation details. The correct functionality of this model should be verified before moving to the next step.

1.1.2 Datapath Desgin

Next the datapath and control logic are developed. Without specifying the physical implementation details, the components such as combinational logic and registers and their connections are designed [1]. Datapath is the combinational and sequential logic which performs the required task. Control logic on the other hand, as its name suggests, is responsible for timely steering of the data within datapath, for example by controlling enable signals or switching multiplexers. The output of this stage is normally a code in a hardware description language (HDL) such as Verilog [12] or VHDL [1] which captures the design at the Register Transfer Level (RTL).

1.1.3 Logic Desgin

At this stage the RTL code gets converted to a netlist which is a more detailed version of the design consisting of logic gates, flip-flops and the way they are connected together. This conversion is called “logic synthesis” which translates a behavioral HDL

code to a netlist of library cells. Other operations within this step are logic optimization, state machine synthesis, datapath optimization and power or area optimization [3].

1.1.4 Physical Design

In the physical domain a gate or a flip-flop consists of a number of transistor with a particular layout. Wires will also have a layout of the corresponding metal layer within the chip. In the physical design step, these masks or layouts are generated which are suitable for fabrication. Thanks to the progress made in the EDA industry, for digital IC design this step has been highly automated, with little or no human involvement necessary. Placement, floorplanning and routing are the tasks required for physical design [3].

1.1.5 Manufacturing

Manufacturing a chip requires many steps in which the semiconductor is shaped and built to match the desired circuit. In a semiconductor fabrication facility the physical layouts are printed as masks and used for several required steps such as deposition, etching, etc[13]. Eventually the silicon wafers with the designed circuitry imprinted on them are fabricated, cut and packaged and most importantly tested to validate the correct functionality.

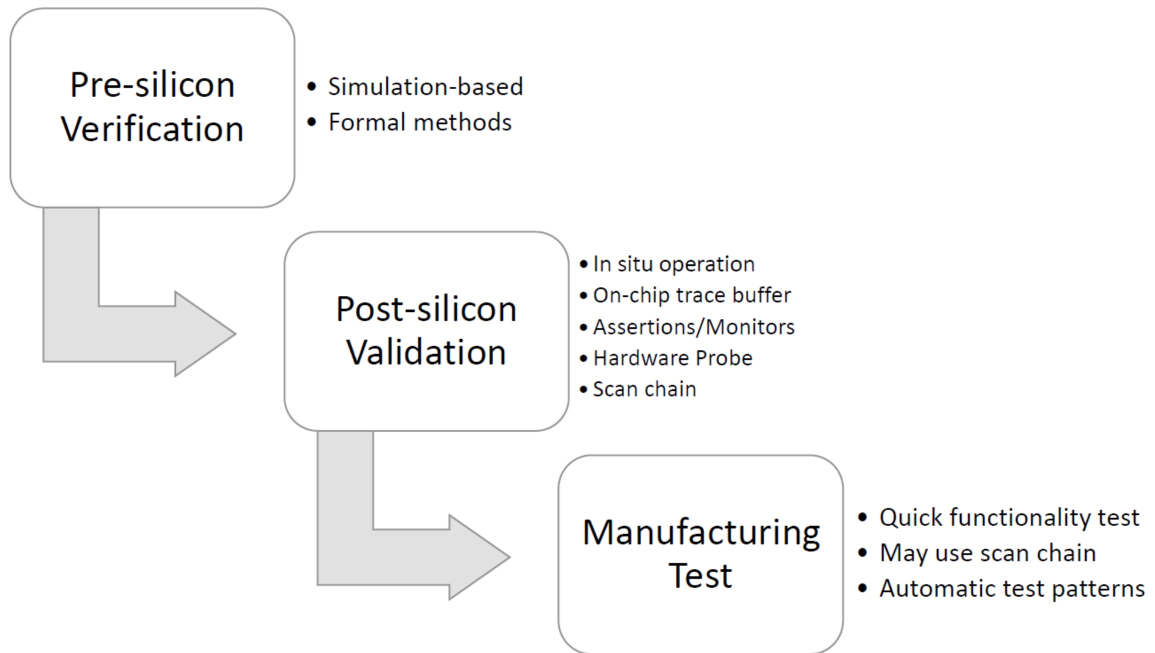


Figure 1.2: Test Flow of Digital Integrated Circuits.

1.2 Test Methodology

A design needs to be tested on multiple stages during the design flow to make sure that the desired functionality is attained. In general VLSI tests fall into 3 categories, depending on their objective and the phase in which tests are run.

1.2.1 Pre-Silicon Verification

These tests, also known as *"functionality test"* or *"logic verification"*, are performed before fabrication while the design is still in the first few stages to make sure that the design performs its functionality. Industry studies show that logic verification has become a major contributor to the total amount of product development time [14]. For example, for a 64-bit adder, one needs to apply many different numbers to the

circuit under test (CUT) and observe and verify its output. Theoretically, one might want to perform the addition for all the possible input combinations, which is most likely practically infeasible as will be discussed later. This can be done through a logic simulator that is capable of running a testbench on the RTL code. Since the number of possible input patterns (2^{128}) is beyond any practical limitation [1], this approach is hardly used in practice. Due to this inherent limitation of simulation, alternative methods such as various types of coverage metrics and application of constraint random stimuli is often used [15]. Code coverage [16], event coverage [17] and state-machine coverage [18] are a few these metrics, which basically monitor the extent of exploration of various parts of code or other criteria while the CUT is being simulated [19].

We may also benefit from the performance gained by using "Emulation" platforms with an FPGA, which can perform a thousand times faster than a what software simulators are able to run [20]. Additional online hardware monitors or property checkers can also be integrated in an FPGA-based test platform which may not be practical to include in the final product[3].

Another approach in pre-silicon verification is through formal methods where a more theoretical approach is used to prove certain properties within the design. A formal method explores the logical equivalence of two seemingly different, supposedly the same designs and provides mathematical proof for their equivalence (or counter example for their difference) [21, 22].

The above mentioned comparison can be applied on two different descriptions

[1]Assuming 2^{128} input patterns are applied at a rate of $2GHz \approx 2^{31}$ samples per second, we need 2^{97} or $\sim 10^{29}$ years to apply all the patterns.

Table 1.1: Comparison between simulation-based and formal methods

	Pros	Cons
Simulation	• Used extensively to find majority of functional bugs • Existence of constrained random stimuli • Existence of many coverage metrics • Almost full observability and controllability over signals in the design	• No guarantee to find all design errors • Slow to perform system wide • Coverage metrics only as effective as the extent of the test vectors
Formal	• Provides mathematical proof	• Lack of scalability to large design blocks • Tools still maturing [3] • Undermined by incomplete reference model

of the design such as a behavioral RTL or a gate-level netlist, or even a set of requirements explained in a high-level language. These specifications could be in a high-level programming language, a system-level modeling language (e.g. SystemC [11]), a hardware description language (e.g. Verilog [12], SystemVerilog [23] or VHDL [1]), a list of input/output patterns, or even a list of requirements in plain human language.

Both simulation based and formal methods have their advantages and drawbacks as summarized in table 1.1. For example the quality of simulation tests using any form of coverage metric highly depends on the test vectors that are applied to the CUT. Similarly, while the existence of a formal proof for equivalence of a design model and a reference model seems very useful, if there are incomplete specifications based on which a reference model is designed, the proof of correctness is undermined.

1.2.2 Post-Silicon Validation

The second type of testsare run on the first batch of fabricated silicon prototypes. Once the logic verification concluded that the design is bug-free the physical design is sent for fabrication. Surveys show that the majority of the first silicon prototypes are still carrying design errors that have escaped pre-silicon verification steps [24]. There are many major differences between pre- and post-silicon tests as will be discussed here.

First major difference is the speed at which tests can be run. Simulations are inherently 5 to 7 orders of magnitude slower than actual chip [25]. When installed on a prototype board an actual circuit can be clocked in the gigahertz range such that a few seconds to minutes of in situ operation leads to hundreds of billions of clock cycles. It might take several years to decades to simulate the same sequence of events.

Another major difference, this time in favor of pre-silicon verification, which in turn makes silicon debug a challenge, is the degree of visibility and controllability of the design in a simulation environment. While a simulator can show the value of any signal within the design at any given time, in the post-silicon domain there are physical limitations as to how many signals we can probe. Main bottleneck is the number of input/output (IO) pins that a single chip can have. Another limitation rises from the amount of data that is generated in real time. Consider the same simple 64-bit adder mentioned previously, assuming that it is clocked at 1GHz, which can generate 8 gigabytes of output every second. Transferring the data out of a chip and storing such a huge amount of information on-the-fly for such a simple circuit is practically infeasible. Alternatively, what we can do in practice is one or more of the

following:

1. Collect only a small portion of these bits and record them onto an on-chip memory, also known as a trace buffer and download it offline, when the functional clock is stopped. This trace buffer is usually a circular FIFO which is used to keep a history of events. This methodology is discussed in further details in this dissertation.

2. Use scan-chains which are built-in shift register structures embedded in a digital design. One can observe the state of internal signals by serially scanning out the information stored in the scan-chain. It is also possible to control the state of these flip-flops by feeding in new values that will overwrite the internal state. Obviously all of this can be done only when the original clock is stopped and circuit is switched to "test mode" in which flip-flops are multiplexed to join the chain rather than normal operation[26].

3. Generate an on-chip signature from a selection of state registers, most likely by feeding them to a Linear Feedback Shift Register (LFSR) [27]. Eventually one should compare this signature with a previously computed value to confirm correct operation. This method is useful for designs with built-in self-test (BIST) [28, 29, 30].

4. Embed hardware property checkers, also known as hardware assertions, which integrate different nets from a design and checks for specific patterns. If the input pattern matches an illegal sequence, the assertion gets violated and an event of interest is therefore detected [31, 32, 33, 34].

And finally, what distinguishes post-silicon validation the most is the presence of many electrical and physical effects in the silicon prototype. These physical phenomena may be very difficult to model in simulation, regardless of the available processing power. Effects such as high or low temperature, multiple clock domain crossings or asynchronous events are very difficult to predict, model and even reproduce.

After a bug is observed in the post-silicon validation phase, there are certain steps needed to be taken in order to fix the problem [35].

1. First step after detection of a bug is to *localize the problem* to a particular module or submodule. Reproducing the problem and how easy it is to repeat and stimulate a particular bug can shrink the time required for this step. We need to answer this question: Under what conditions or workload does the bug get activated?

2. *Finding the root cause* is the next step in which we need to find the physical reason behind the bug. For example, is it a particular path in the circuit that works slower/faster than expected under certain physical conditions, such as voltage droop or extreme temperature?

3. Last step is to *fix the bug*, as well as deciding how to fix it based on the cost and depth of the bug. Generating new fabrication masks is usually the most expensive solution for when the design or at least part of it gets updated.

1.2.3 Manufacturing Test

Once all the pre and post-silicon test have passed and a chip is fabricatedevery single chip undergoes a final test to make sure that there is no physical defect that affects

that particular piece of silicon. This is called *manufacturing test*, or merely *"test"* within electronic test community [36] . There are many manufacturing defects that can potentially affect both the functionality and the lifetime of a chip. Some examples are [3]:

- Metal to metal shorts
- Disconnected wires
- Damaged Vias
- Shorts through the thin oxide of gate to the substrate

These defects can cause nets to short to ground, power, or other nets, or potentially lead to disconnected nets. Manufacturing test is in charge of verifying that all the internal gates and input/output pins are functioning as expected. Exercising all gates within a design, and generating test vectors that conduct such a test is a burden facilitated by systematic approaches such as Automatic Test Pattern Generation tools (ATPG).

Finding the maximum operational speed of a chip or the minimum voltage for such a speed is another task that is carried out during the manufacturing test. Figure 1.3 shows an example of such speed test for an Intel ATOM processor [2]. As shown in this figure the chip operates correctly at a range of frequencies between 1.25 and 2.5 GHz, with a voltage range of 0.8 to 1.2 volts. In this shmoo diagram the "*" character represent correct behavior under test and other characters represent different types of failure.

The major differences between manufacturing test and post-silicon validation are:

- Post-silicon validation is applied only on a limited number of silicon samples, mainly the first batch of prototypes, to detect any design errors that have escaped from the implementation phase to fabrication.

- Manufacturing test is applied to every fabricated integrated circuit to make sure they are free of fabrication-induced defects.

- Validation needs to pass before committing to mass production of a chip while manufacturing test is done in parallel with it.

- Validation of prototypes may take weeks to months while the manufacturing test takes only seconds per chip. A failure in post-silicon validation needs to be debugged until the root cause of the problem is detected and fixed. A failed chip during the manufacturing test may just be discarded, unless it is suspected that there is a systematic defect that needs to be addressed.

It is worth mentioning that certain types of fabrication defects that manifest themselves as slow paths can be caught through delay testing, however, they are not sufficient to capture subtle design problems that cause consistent failures on application boards under special operating conditions [37].

1.3 Thesis Organization

Before discussing the contributions in further detail, in chapter 2 we will introduce several other works in the field of post-silicon validation that are related to the topics of this thesis.

The three main contributions will be covered in Chapters 3, 4 and 5:

Figure 1.3: Shmoo Graph of Intel's ATOM processor. [*] characters represent a pass, other characters show various types of failures [2].

- **Chapter 3:** Automatic Bit-flip Detection Test Platform

  Details about the automatic test platform such as using Boolean satisfiability (SAT) solvers to detect bit-flips and root cause analysis will be covered. Other topics covered in this chapter include the simulation platform, how to inject bit-flips, details on translating a circuit to Boolean SAT problem, and extracting the list of suspect flip-flops from the unsatifiability core (core of UNSAT) generated by a solver.

- **Chapter 4:** Trace Signal Selection for Bit-flip Detection

  Details of the proposed algorithm for trace signals selection will be covered in this chapter. "Bit Flip Detectability" metric will be introduced which is at the core of our heuristic algorithm.

- **Chapter 5:** Joint Trace and Assertion Selection for Bit-flip Detection

  Various strategies for joint selection of assertion checkers and trace signals will be discussed which are an essential part of the work from a research stand point.

- **Chapter 6:** Conclusion and Future Work

A summary of the entire dissertation is provided as well as some ideas for future work.

Each chapter includes results for several experiments as well as the challenges for each contribution.

Chapter 2

Background and Related Works

As mentioned in the previous chapter, there are three different categories of tests, each applied at a certain stage of design and production of digital integrated circuits:

- Pre-Silicon Verification
- Post-Silicon Validation
- Manufacturing Test

Each type was also briefly introduced in the previous chapter. Since the contributions of this dissertation fall within the realm of *post-silicon validation*, more details on it will be provided. This chapter begins by emphasizing the importance of the post-silicon validation, explaining the challenges and some solutions.

2.1 Why Post-Silicon Validation?

Integrated circuit manufacturers need to invest a considerable amount of resources into post-silicon validation to make sure that the fabricated prototypes meet the presumed specifications, before committing to mass production of a design. This is because pre-silicon methods are compute-intensive, and hence insufficient to identify all the design errors (or bugs) before tape-out. As a result, there are design bugs that escape the pre-silicon screening and end up in the silicon prototypes. A recent study shows that post-silicon validation is becoming increasingly time consuming and may adversely affect the time-to-market [24]. It also shows that two out of every three silicon prototypes were erroneous, despite the significant investment during the pre-silicon verification. The majority of designs reported in [24] required two to four re-spins before committing to high-volume manufacturing. According to some case studies, the time-to-market is expanding in part due to challenges during post-silicon validation [38].

2.2 Major Challenges

Post-silicon validation has emerged as a challenging task in the implementation cycle due to the following major hurdles.

2.2.1 Controllability and Observability

First major challenge lies in the limited observability and controllability of the internal signals. This is in contrast to pre-silicon verification, where any signal from the design can be observed and/or controlled. At the post-silicon stage, observability

is ultimately constrained by the number of input/output (I/O) ports. In order to observe or control a particular signal after fabrication, one needs to have connected the signal to an external pin at design-time. Methods such as data compression, multiplexing or using additional circuitry may be utilized to group several signals, hence observing/controlling more nets, although only up to a certain extent [39].

2.2.2 Simulation and Golden Response

Another challenge stems from the lack of a golden response. During pre-silicon verification we rely on simulation-based methods to inspect the behavior of a design. Simulation is inherently several orders of magnitude slower than the actual circuit [25]. As a result, seconds to minutes of at-speed operation of a fabricated circuit can easily outperform what can be achieved through simulation in days to weeks. It is therefore practically infeasible to generate the golden response of a complex design for non-trivial tasks (for example simulating an hour-long video game during pre-silicon verification).

2.2.3 Reproducibility

Another major challenge while diagnosing a circuit prototype is to reproduce the exact electrical state that leads to a failure. Design practices such as multiple clock and voltage domains, combined with unpredictable events caused by asynchronous inputs, voltage droops and extreme temperatures, may activate a particular design error that is not easily repeatable.

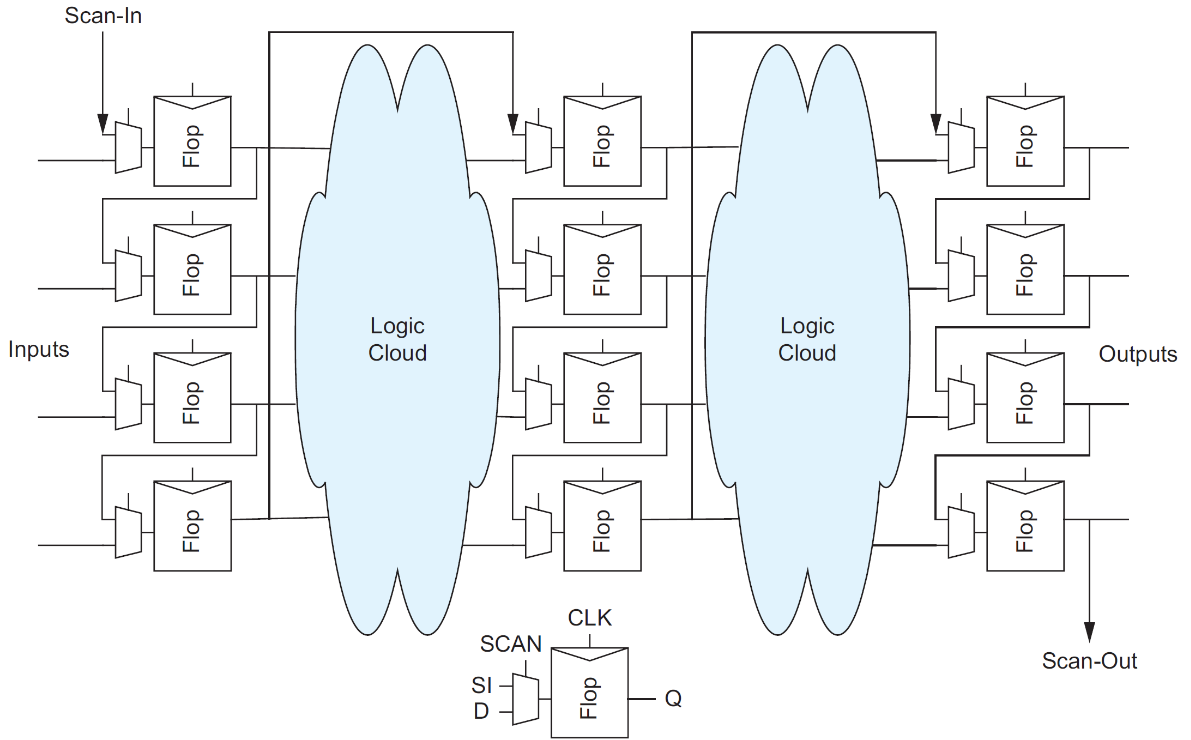


Figure 2.1: Illustration of the serially connected scan chain. Each Scan-Flip-Flop consists of a regular flip-flop attached to a multiplexer that enforces the normal operation mode or a test mode. Reproduced from [3]

2.3 Notable Solutions

2.3.1 Scan-chains

To tackle the limited observability and controllability, scan chains can be leveraged to overwrite and/or read out the state of a digital circuit (scan dumps). In this technique, which is one the most popular Design for Testability (DFT) methods, all the flip-flops which are supposed to become observable or controllable will be replaced with reconfigurable flip-flops or *scan-cells* [40, 41]. Each scan-cell consists of a multiplexer that chooses between the normal operation of the circuit or a test mode.

In the normal operation mode, each flip-flop is driven by its original source net. In the test mode, however, each flip-flop is switched over to the output of another scan-cell to form a chain [26]. Scan-cells operate either in the normal operational mode or test mode. A circuit can work as usual until the test mode is asserted to the scan-cells. In test mode, the value of all the scan-cells, which form a long shift register (or chain), can be serially scanned out, hence their state will be observed. Simultaneously one can serially scan in new bits to overwrite the old state of scan-cells, hence improved controllability.

In its simplest way, in order to scan in/out bits from a scan-chain, first the operational clock source of a circuit needs to be stopped, then the scan-flops are switched to scan mode to form the chain, and finally the scan-flops are clocked as many times as the length of the chain and the output of the chain is sent out to an external device and recorded as a scan dump. This technique has facilitated testing for fabrication defects for many decades, despite the fact that the scan clock may be slower than the operational frequency. Nonetheless, scan dumps provide no information on the sequence of events that have preceded the failing state and one has to know exactly when to stop the circuit execution. As will be explained later, *trace buffers*, which are on-chip memory elements, remedy this limitation since they have the advantage of recording a history of events for tens to hundreds of clock cycles.

2.3.2 On-Chip Stimuli Generation

As summarized by figure 2.2, the scan-chain method helps with both controllability and observability, even though it has its own limitations. In contrast trace-buffers, as will be discussed later, only improve visibility by providing a history of events for

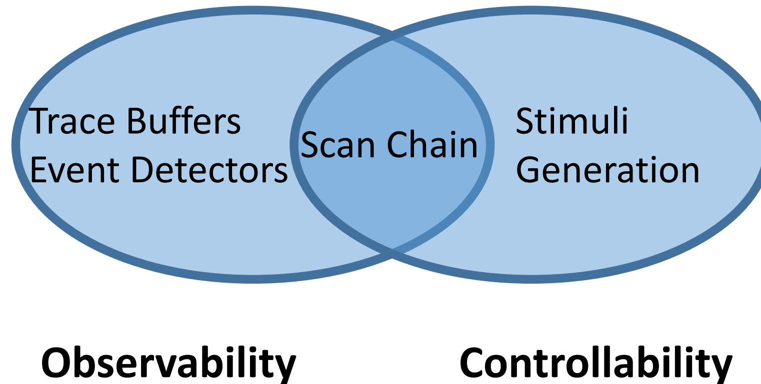


Figure 2.2: Added on-chip hardware can improve controllability and observability. Some methods such as Scan Chains help with both, while others focus only on a specific aspect.

a number of clock cycles. Another approach to improve controllability is to generate real-time stimuli patterns within the chip.

During the post-silicon validation, it is practically infeasible to generate and provide a high volume of real-time test vectors from an external source (presumably at gigahertz rate) due to limited bandwidth and software execution speed. Similar to pre-silicon verification in which a large number of random patterns are applied to the design, in post-silicon there have been attempts to provide such random stimuli. A large number of random vectors increases the chance of detecting bugs [42, 43, 44]. As a result, in order to perform post-silicon validation one can benefit from generating and applying a set of *constrained random stimuli* on-chip and in real-time.

Note that the random stimuli have to be compatible with the design at hand and must match the input requirements. For example a video decoder may accept only very specific sets of input patterns in order to operate according to the specifications. For microprocessors, which are only a specific yet widespread digital circuit, there

are methods to generate a sequence of machine codes to stimulate the DUV with instruction-level templates [44, 45]. There are also other methods that focus on generating constrained random stimuli for generic digital circuits [46, 47, 48].

Since on-chip stimuli generation is beyond the scope of this work, we only introduce the idea from [46, 47, 48] in a brief and rather simplistic manner. As shown in figure 2.3 on-chip stimuli generation relies on pseudo-random generator such as an LFSR to generate a stream of variable bits. Then, these bits are processed together with a mask pattern tailored specifically for the *design under validation*. These patterns can be transferred from the software that make them to the *correction logic* inside the chip. The correction logic then adjusts the pseudo-random bits to the required pattern. The output of the correction logic is expected to be of the correct format suitable for the design under validation. The interested reader can learn more about this aspect of post-silicon validation from [49].

2.3.3 Error Detection

The usually cumbersome process of silicon debug would have become significantly easier if one could either quickly reproduce the erroneous behavior of a system or detect the error immediately or shortly after it has occurred. Unfortunately, in practice most debug sessions often take a long time since neither of these two conditions are guaranteed to hold. In the former scenario (reproducibility) one could potentially repeat many debug sessions in a relatively short time and collect some information in every turn (such as scan dumps, trace buffers or simply by observing I/O) until enough evidence of the root cause is found. In the latter case (quick error detection) the erroneous behavior has less time to propagate and cause more corruption to the

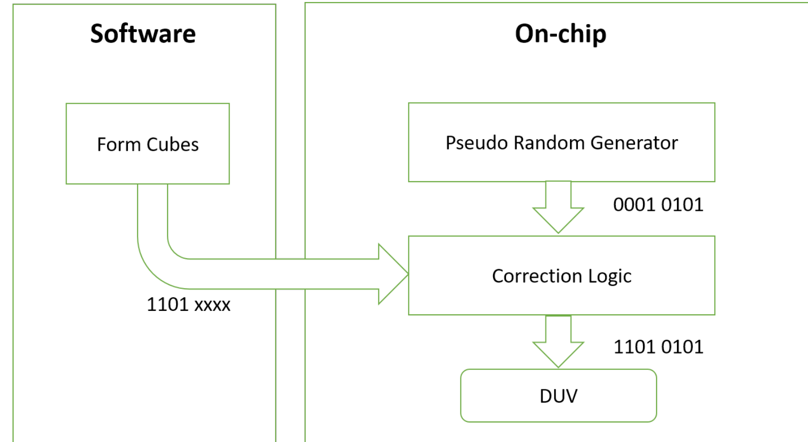


Figure 2.3: The software on the left processed the circuit and generates a set of cubes according to the verification requirements. The cubes contain fixed bits (0/1) and flexible bits(x) that can be randomly filled with either 0 or 1. The cubes are then loaded to a hardware memory. The correction logic receives a pseudo-random sequence of bits and matches it to the received cube, basically filling the x bits with 0/1.

state of the system. Hence, by stopping the system under test immediately after error detection and extracting the state (e.g., scan-dump [26]), it is possible to search for the inconsistencies in a narrower space. For example, using JTAG [1] test access port one can offload the state of the scan-chain [50]. Note that, since scan dumps provide no history of relevant events, if *error detection latency*, which is the time between the bug getting activated until somehow observing its effect, is not short enough, scan-dumps may provide no useful information[51].

In order to reduce error detection latency, additional on-chip circuitry can also be utilized to detect out-of-ordinary events. [52] has proposed an on-chip programmable trigger unit which can detect various events such as level or edge sensitive triggers or a sequence of events. More recently microprocessor-focused methods have also been

[1] Joint Test Action Group. Please refer to IEEE Standard 1149.1 for more information: `https://standards.ieee.org/findstds/standard/1149.1-2013.html`

proposed that basically duplicate the original instructions and regularly compare the outcome of the two sets of operations and expect them to be the same [53, 54].

More importantly, *assertions* which were traditionally only used in software development and pre-silicon verification have found their way to post-silicon [31, 55, 56, 57]. This is owing to two main factors:

- **Hardware Synthesis of Assertions:** A designer can translate an assertion into hardware by forming a finite state machine (FSM) that detects a certain illegal pattern on a number of signals. In recent years researchers have developed several automatic methods to map assertions to hardware automatically [58, 59, 60, 61].

- **Automatic Assertion Generation:** Traditionally a designer had to manually write assertions based on the requirements/specifications. More recently, systematic methods to extract/generate assertions for a given design have been developed [62, 63, 64].

In this work we benefit from automatically generated hardware assertions without diving into their implementation details. As will be explained in chapter 5, a failed assertion will provide the debug procedure with very useful information that can narrow down the list of suspects and improve the accuracy of the root cause analysis.

2.3.4 On-chip Trace Buffers

To collect real-time information from within the circuit, it is common practice to use on-chip memories as trace buffers that record a history of a subset of signals. This extra information is critical during root cause analysis of the failure because the

history from the trace buffer can give unique insights into what lead to the observed failing behavior. Nevertheless, the subset of signals that are to be recorded needs to be known at design time. One has to decide in advance which signals will provide more information during a debug session, for example by improving the overall visibility. A designer may use the inner knowledge of a circuit's structure and decide which wires to tap for debug purposes. Since manual selection is cumbersome, especially for circuit blocks with tens or hundreds of signals of interest, there is a need for systematic approaches for trace signal selection. Also, there may be non-obvious logical relations among a group of signals that are overseen by the designer, in which case an algorithmic approach will be more effective.

Several studies have addressed the question of *"which signals to trace?"*. Initially the work from [65] proposed a metric to quantify the effectiveness of tracing a set of signals called *restoration ratio*, which is the number of bits restored divided by the actual number of bits collected. This work was followed by many other approaches, which are either analytical, simulation-based or hybrid, that either improve the restoration ratio (by introducing a new selection algorithm) or decrease the runtime of the algorithm from minutes to seconds (e.g., [66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76]). Intuitively restoration-based methods justify their effectiveness for functional bugs where the known logical values can safely propagate to other signals (spatial propagation) and other clock cycles (temporal propagation). However, this metric does not provide a one-fit-all solution when tracing is used for other objectives.

Several of the selection algorithms are structure-based, which means that they make decisions based on the connections among logic gates. For example [66] defined

a probability-based method to improve gate-level visibility trace signal selection algorithm which was able to restore more missing bits compared to the original work in [65]. Similarly in [67] logic implication of every gate over other nets is used as the basis for choosing trace signals with more influence.

The main criteria in which these selection algorithms could be compared are either the quality of the selected signals measured mostly by restoration ratio, or the runtime of selection procedure. The methods in the literature fall into metric-based or simulations based methods. Metric-based methods mostly use probability (such as [70]) or their own defined metric (such as debug-difficulty in [72]). In [77], assertion coverage is proposed as an alternative metric for trace signal selection in post-silicon validation. Simulation-based methods may potentially have a better selection quality, since they rely on real or synthesized input vectors to stimulate the circuit and observe the logic behavior and choose signals accordingly. The higher quality of selection quality comes at the cost of high simulation time. [74] and [76] manage to keep the simulation time scalable for circuits of upto 50000 nets while using methods such as machine learning or ILP (Integer Linear Programming). Other methods employ various novelties to achieve their objective: for example [71] performs a hybrid selection based on both simulation and probability. [68] first determines critical state elements and then performs selection to achieve a better selection quality. [69] selects trace signals as well as scan-cells that together can potentially restore many bits. For logic circuits that have multiple modes of operation, [75] introduces a method that judiciously select groups of signals for each mode of operation.

The methods mentioned above have a common objective which is to improve the *restoration ratio* and ultimately restore more unknown bits within a debug session,

in the absence of any bit-flips. In the presence of bit-flips, caused by electrical bugs, a logical state might be restored to a wrong state due to the presence of a logical inconsistency. Inefficacy of restoration based trace selection has been observed also by others, such as [78] where it was shown that a random selection is more effective than the above mentioned restoration-based selections, when bit-flip detection is the objective. This will be discussed in more detail in the following section.

2.3.5 On-Chip Tracing in the Presence of Electrical Bugs

Trace buffers can be quite helpful when dealing with "functional bugs" since for this type of bugs logical consistency is preserved. For example if a digital designer implements a 1's complement negation when in fact 2's complement has been required, all the silicon prototypes will be affected by such a mistake. Collecting a number of bits inside a trace buffer and post processing them could potentially reveal the wrong outcome as well as the underlying reason. In this scenario, since there is no logical inconsistency the restored bits for unknown signals are meaningful. If, however, there was a cross-talk between two signals within a design, where one could affect the other such that it would flip its Boolean value, presumably at a certain voltage or frequency, then we would be dealing with an "electrical bug".

Electrical bugs, when activated, affect a digital circuit by undesirably flipping a bit. A few examples of these bugs are those related to corner cases due to process variations, temperature and voltage-sensitive signal races or asynchrony from multiple clock domains or I/Os, all of which are not easily reproducible [79]. Due to the sophisticated nature of electrical bugs, they can potentially affect some or all of the prototypes. In the presence of these bugs that commonly result in bit-flips in the logic

domain, restoration-based methods are insufficient. This is because when restoration ratio is used as a metric, the objective is to select trace signals that eventually restore more bits in the circuit over multiple clock cycles. Such algorithm does not take into account: (1) the occurrence of a bit-flip and how they change the behavior of the circuit; (2) bug localization, i.e., in which flip-flop and in which clock cycle the bit-flip occurred. As a result, in this work we address the following problem: *"which signals should we trace in order to detect and localize bit-flips?"*.

An example of a systematic solution to facilitate bug localization during post-silicon validation is Instruction Foot-print Recording and Analysis (IFRA) [80], which was designed to be applied to microprocessors. An improvement to IFRA was introduced in [81], nevertheless it is not clear how these methods can be applied to generic digital logic blocks that are common in system-on-a-chip (SOC) devices. In chapters 3 and 4 we present an automated trace buffer-based solution for searching for bit-flips in generic logic blocks. Our approach relies on *Boolean Satisfiability* (SAT) formulation. Boolean SAT has been extensively studied in theoretical computer science and used in electronic design automation (EDA) applications, such as pre-silicon verification [82], manufacturing test [83] and model checking [84].

A Boolean SAT instance is either satisfiable or unsatisfiable. For an *unsatisfiable* Boolean SAT instance a subset of clauses within the problem cannot be satisfied simultaneously. As discussed later in this thesis, this particular subset, also known as *"core of UNSAT"*, provides critical information that points out to the logical inconsistency hidden in the trace buffer. By analyzing this core we can generate a list of suspect nets where the bit-flip has likely occurred. The work presented in [85] has considered debugging on the gate level by restricting the SAT instance to

the failure trace and the correct output values. Similar to our work the constrained problem is passed to a SAT solver to extract the core of UNSAT. However multiple counterexamples are assumed which is in contrast with debugging transient bit-flips. This is because experiments are not easily reproducible. Another major difference between error localization in our work and [85] is that in our approach the correct output values are not assumed to be available. More recently, the approach presented in [86] uses an iterative window-sliding procedure, which is claimed to be scalable for large circuits by making the size of the sliding window adjustable. They identify a small set of fault candidates, however, their assumptions are different from our work: (1) primary input/outputs (I/Os) are assumed to be traceable from the beginning of the debug experiment. Since subtle design errors might take days to weeks to manifest themselves [87], the assumption from [86] cannot be applied to such problems due to excessive storage requirements. In contrast, we make no assumption about recording the I/Os and we rely only on the data recorded in the on-chip trace buffers. (2) The work from [86] focuses only on trace analysis when a *random* subset of signals are traced. In contrast, the main contribution of our work from chapter 4 is to define new metrics that can guide trace signal selection algorithms that will assist with bit-flip detection and localization. The key insight lies in the fact that trace data collected during a debug experiment should imply opposite values in the flip-flop where the bit-flip has occurred.

2.3.6 The Debug Process and the Scope of This Work

As depicted in figure 2.4 the debug process using trace collection consists of several steps. These steps are divided into two major phases: pre-silicon and post-silicon.

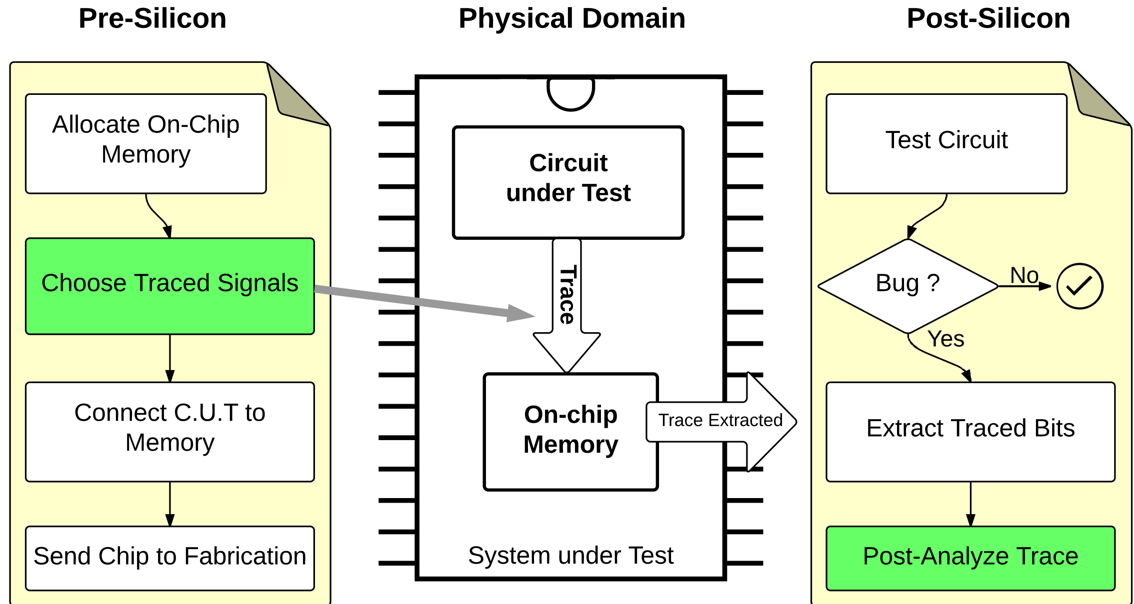


Figure 2.4: Pre-silicon and post-silicon tasks for trace-based debugging.

Only the highlighted item falls within the scope of this work, as explained below. In the pre-silicon phase, we assume that a circuit is finalized and ready to be manufactured. Then the trace collection circuitry is added to facilitate the post-silicon debugging. We first allocate on-chip memory for trace collection. This can be limited by chip area and the cost of on-chip memory. Then we need to choose the trace signals based the memory budget from the first step. Afterwards the circuit has to be modified to connect the traced signals to the memory. Optionally, one can include several event detectors or hardware assertion units to monitor various events or trigger data acquisitions. Finally the chip is sent for manufacturing.

For the post-silicon phase, the manufactured circuit is exercised until an error is observed, at which point the system is halted and the collected trace is extracted out of the on-chip memory. This information needs to be processed so that we find

the root cause of the problem. The three major contributions of this dissertation are listed below. We should note that other tasks during post-silicon debugging such as event detection [52], or assertion generation [62, 63, 64] are critical steps that will also influence the analysis of the failing traces caused by bit-flips, however they are beyond the scope of this investigation.

2.4 Overview of Contributions in this Thesis

As mentioned before automated solutions such as CAD tools for verification/validation are an essential part of the VLSI design and test methodology. Below we enumerate the contributions of this thesis to facilitate post-silicon validation in different stages.

1. Automatic detection of bit-flips and root cause analysis through analysis of the failing traces, in Chapter 3,
2. Trace signal selection to maximize bit-flip detection, in Chapter 4,
3. Concurrent selection of trace signals and assertion checkers to maximize bit-flip-detection, in Chapter 5.

2.4.1 Automatic Detection of Bit-Flips

Chapter 3 discusses a test platform in which bit-flips, which are presumably unexpected and undesired change of bits as a result of an electrical bug, are detected in an automated process.

In order to do so, several software modules were developed that would run the following steps:

1. Receive a circuit in terms of a netlist of gates and flip-flops.
2. Modify the circuit such that a bit-flip can be injected in a simulation platform at any given time to any given flip-flop.
3. Run simulation and inject bit-flips to all flip-flops at several distinct time-steps.
4. Record the values of all or several flip-flops before and after the injection and store them for post-processing.
5. Convert the netlist to a Boolean representation of the circuit, while strictly keeping the logical relations of the gates in the new representation.
6. Convert the recorded bits of a number of flip-flops, also known as the trace bits within a trace buffer, to the same mathematical format.
7. Match the two mathematical representation and check if they are compatible or alternatively they are inconsistent.

In summary the input and output of this contribution is:

INPUT
1. A circuit,
2. List of flip-flops assumed to be traced,
3. Which flip-flop to inject a bit-flip into,
4. What time to inject that bit-flip.

OUTPUT
1. Test outcome which is either "bit-flip detected" or "bit-flip NOT detected",
2. If the bit-flip is not detected, there is not much information that could be provided at this stage. If, however, the bit-flip was detected additional information such as list of suspect flip-flops, or,

3. The range of time-stamps in which an inconsistency is detected.

The information provided as output are helpful for bug localization for post-silicon validation as mentioned in the previous section about validation.

2.4.2 Trace Signal Selection

Chapter 4 answers the next big question: "which signals to trace to detect most ratio of bit-flips?" In the previous contribution, we assumed that the list of traced signals is somehow given. Such a list can come from the designer of the circuit or can be a random subset of all the flip-flops. Alternatively, as intuition suggests, one could develop an algorithm to systematically select a subset of flip-flops.

The objective is to precess a circuit and, given a certain budget, provide a list of more influential flip-flops which can help uncover more bit-flips. Note that, there are many "trace signal selection" algorithms already in the literature, however they have their own different objective such as maximizing restoration or detecting functional bugs, rather than bit-flips or electrical bugs.

The big picture of our proposed algorithm is as follows:

1. Receive a circuit in terms of a netlist of gates and flip-flops.
2. Model the circuit's connection with a probabilistic model.
3. Mark various flip-flops as "traced" and observe their effect on the probability of detecting bit-flips on other flip-flops.
4. Choose the flip-flop which causes the maximum detection probability.
5. Repeat above for as many times as the number of required traced flip-flop, selecting one in each cycle.

6. Provide the final list of selected flip-flops.

In summary the input and output of this contribution are:

INPUT
1. A circuit,
2. Trace budget or the number of flip-flops to be selected.

OUTPUT
1. List of selected flip-flops.

Together with the first contribution, the effectiveness of various trace selections can be quantified. Different selections may come from either a random selection or a selection with a modified version of our proposed algorithm. This will be discussed in detail in chapters 4 and 5.

2.4.3 Concurrent Trace and Assertion Selection

Hardware assertions can be used to detect irregular events within a digital circuit. They can be utilized as an event detector or a means of compressing several nets into a single digital flag. In the previous sections we selected a group of traced flip-flops with the aim to detect as many bit-flips as possible. Next we decided to integrate a number of hardware assertion checkers to further improve bit-flip detection.

In our proposed algorithm the following steps are achieved in addition to all the steps in the trace-signal selection algorithm:

1. A pool of hardware assertions is received.
2. The effects of each hardware assertion is first quantified in terms of the bit-flips it detects. Also the cost of each assertion, in terms of wire count or area overhead is calculated.

3. Mark various flip-flops as traced and measure the effect in estimated bit-flip detectability. Also integrate individual hardware assertion and measure the same effect.

4. Choose the "traced flip-flops" or "assertion-checkers" which improve bit-flip detectability.

5. Repeat the above steps according to the trace and assertion budget.

6. Provide the list of selected flip-flops and assertions.

In summary the input and output of this contribution are:

INPUT
1. A circuit,
2. Trace budget or the number of flip-flops to be selected,
3. Wire count of the assertions or the total number of assertions.

OUTPUT
1. List of selected flip-flops,
2. List of selected assertions.

In this task we assume that a pool of assertions is provided from an external source. As discussed in [88] there are a number of ways to automatically generate thousands of assertions for any given circuit. Assertion generation is beyond the scope of this work. Our proposed method only selects a subset of the provided pool aimed at detecting as many bit-flips as possible.

In this chapter, several major challenges of post-silicon validation were discussed, as well as a few of the most common solutions. On-chip trace buffers and their application towards silicon debug were introduced. In the next chapter an automated bit-flip detection method is proposed.

Chapter 3

Satisfiability-Based Test Platform

3.1 Background

In the previous chapters we addressed the challenges of the tests required for detecting and debugging during post-silicon validation. It was explained that there is a need to establish a systematic and automated process to deal with the complexity of such tests. The use of on-chip trace buffers was briefly introduced and will be elaborated more thoroughly here. As the main topic of this chapter, we present a computational approach to analyze the failing traces caused by electrically-induced bit-flips.

Here we assume that there is a digital circuit together with a trace buffer that records samples of some of the internal circuitry. Due to the nature of the failing mechanism, we do not assume that the failing trace is reproducible and therefore we rely exclusively on embedded logic analysis for data acquisition. It is important to note that despite significant research in the fault-tolerant community on methods to recover from bit-flips, there are no systematic approaches to root-cause them, which is a critical challenge during post-silicon validation. Our method relies on the

Boolean satisfiability problem (SAT) that has been extensively studied in theoretical computer science, with many practical applications ranging from artificial intelligence to electronic design automation. For example, SAT-solvers have been used in pre-silicon verification [82] and manufacturing test [83]. However, a specific feature of SAT solvers, i.e., the core of UNSAT (as elaborated later in this chapter), provides critical information that is beneficial to understanding the failing traces during post-silicon validation. This feature has motivated our approach and our results indicate that this is a direction worth pursuing as a generic solution to narrow down a suspect list of flip-flops and timeslots (where and when bit-flips have occurred) within the failing trace.

In the following an overview of the trace-based debug process is given, followed by a few motivational examples, the assumptions and the nomenclature and the SAT formulation for post-silicon debugging.

3.1.1 The Scope of This Chapter

Figure 2.4 showed the entire debug process using trace collection which consists of several steps. As mentioned before these steps are divided into two major phases: pre-silicon and post-silicon. In this chapter we focus only on the last step which is to post-analyze the recorded trace. This step is highlighted in figure 3.1. Here we assume that a digital circuit containing an on-chip trace memory has been fabricated and is under post-silicon validation. The circuit is exercised until an error is observed, at which point the system is halted and the collected trace is extracted out of the on-chip memory. This information needs to be processed so that we find the root cause of the problem. The ideal outcome of such post-analysis is to find out exactly at which clock

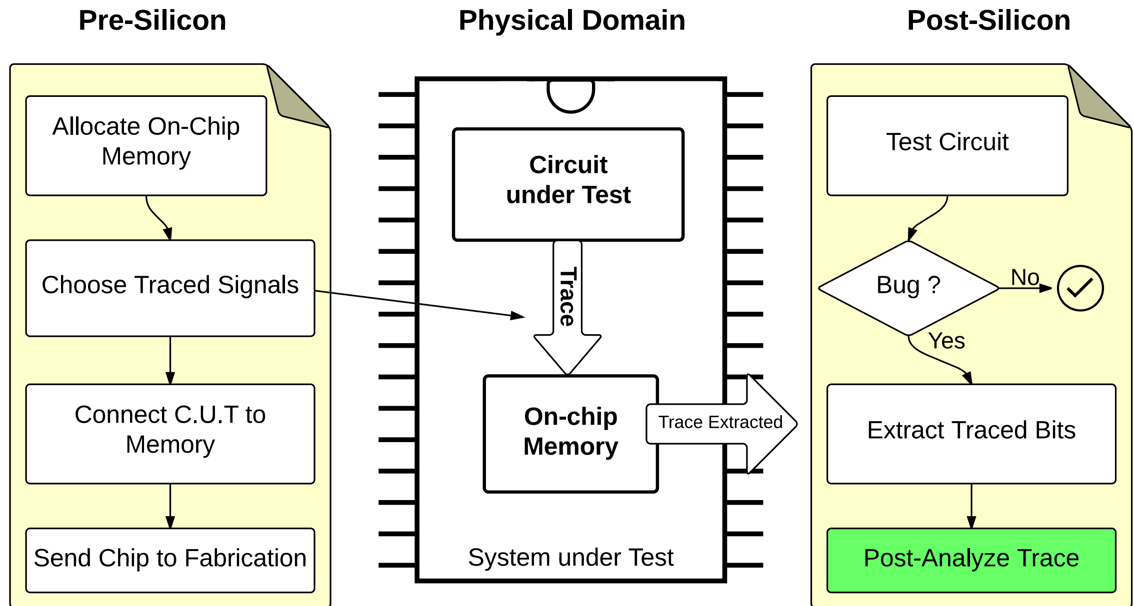


Figure 3.1: Pre-silicon and post-silicon tasks for trace-based debugging. Last step is to process the collected trace and find out the root cause of the bit-flip.

cycle and on which circuit's net a bit-flip has occurred. A more realistic outcome is that some of the bit-flips are never detected and for those that are detected, a list of suspect nets on several clock cycles are generated (hence less than ideal accuracy).

3.1.2 Motivational Examples

Figure 3.2 shows two versions of a simple circuit. The value of `Q` outputs of flip-flops are written next to each pin. The circuit on the left shows consistent values since the logical `AND` of $\{0, 1\}$ is 0 and the logical `OR` of the same inputs is 1. The right half of this figure, however, shows that the `OR` of two inputs is observed as zero while the `AND` is observed as one. This is a logical contradiction because for no values of `A` and `B`, we can have such results for `C` and `D`. Table 3.1 represents the same values

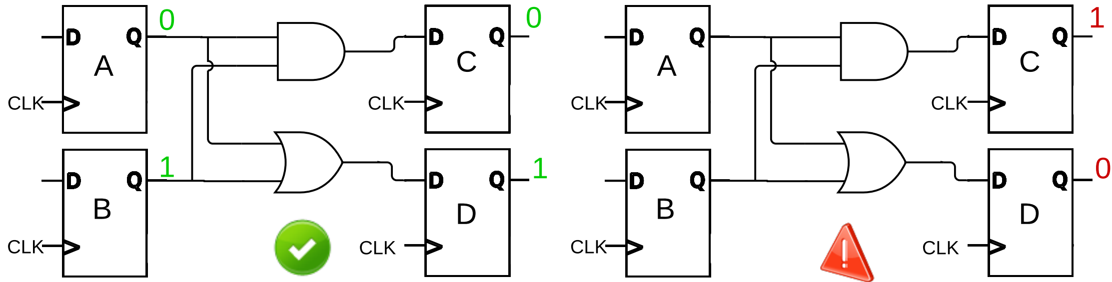


Figure 3.2: An example circuit with consistent values on the left-hand side and an illegal combination on the right-hand side.

Table 3.1: Sample trace for the circuit from figure 3.2 and the corresponding SAT conditions

	A	B	C	D
T	**0**	**1**		
T+1			**0**	**1**

	A	B	C	D
T	?	?		
T+1			**1**☒	**0**

$$(\overline{A_T}) \cdot (B_T) \cdot$$
$$(\overline{C_{T+1}}) \cdot (D_{T+1}) = 1$$

$$(C_{T+1}) \cdot (\overline{D_{T+1}}) = 1$$

of figure 3.2 in a tabular fashion. It consists of two rows representing the two clock cycles that the logic values are extracted form a circuit. Some cells are left empty only for simplicity. Clock cycles T and $T+1$ are two consecutive cycles for which the trace has been available. We will soon show how these contradictions are detected and reported using a SAT solver, after converting the circuit and its trace to a SAT formulation. This table also shows some SAT conditions which are discussed later.

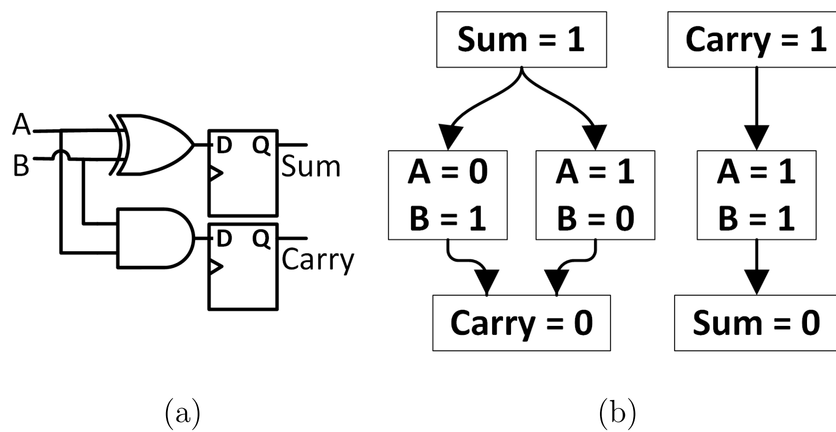
Figure 3.3 is another example to illustrate how logic inconsistencies caused by bit-flips can be detected automatically. We assume that the output {`Carry, Sum`} (or equivalently {`C, S`})is observed to be {`1,1`}. While this inconsistency may be obvious for the reader, we have to provide an automated way to detect non-obvious

illegal states. A step-by-step approach is depicted in figure 3.3 where we try to find the state for the rest of the circuit that leads to the observed pattern. Arbitrarily we start by propagating the output of the `XOR` gate to its inputs. The inputs have to be either {`01`} or {`10`}. Taking any of the paths on this search tree results in `Carry=0`. Alternatively we could begin by back-propagating the `1` value on the `AND` gate, which forces the inputs to only {`11`}. This time we infer `Sum=0`, which is again inconsistent with the collected trace.

The search explained above is implicit in a SAT solver. Figure 3.3(c) shows the procedure of formulating a SAT instance by translating circuit nodes to Boolean variables and using the collected trace data as additional constraints captured through single-literal clauses. If the SAT instance is unsatisfiable (UNSAT), the core of UNSAT will return only the clauses that are not simultaneously satisfiable. By post-processing this core of UNSAT we can narrow down the suspect list of flip-flops and the clock cycles where the bit-flip that caused the logic inconsistencies might have occurred.

3.1.3 Assumptions and Nomenclature

We assume that we are dealing with a digital circuit that is composed of only basic logic gates (i.e. `NOT, AND, OR`, `XOR` and their inverted) and D-type flip-flops. There is also a limited amount of on-chip memory available for storing a history of values from a subset of these flip-flops. This is called a "*trace memory*" and it works in a first-in-first-out fashion with depth of n, a circular buffer in which only the last n samples are recorded and the newest sample will overwrite a sample from n clock cycles earlier. Every flop is either *traced*, meaning that its values is recorded onto the



(a) (b)

$$(a + \overline{c}) \cdot (b + \overline{c}) \cdot (\overline{a} + \overline{b} + c) \cdot$$
$$(a + b + \overline{s}) \cdot (\overline{a} + b + s) \cdot (a + \overline{b} + s) \cdot (\overline{a} + \overline{b} + \overline{s}) \cdot$$
$$(s) \cdot (c) = 1$$

(c)

Figure 3.3: (a) Carry and Sum are both 1. (b) Unsuccessful attempt to find A and B due to logic inconsistency. (c) Equivalent SAT problem. First line describes the AND gate, second line the XOR and the third line is derived from the collected trace bits. In conjunctive normal form (CNF) all clauses must conjunctively evaluate to 1.

trace memory, or *untraced* otherwise. Note that the value of an untraced flop may still be inferred from other traced signals, or stay unknown. We also assume that the circuit has been running for a long time before the traces are extracted out of the memory. The size of the trace memory is w bits wide by n which means that it is large enough to keep a history of w traced flip-flops for the last n cycles.

Table 3.2 illustrates such a trace memory for a hypothetical circuit with 3 traced signals, $w = 3$. In this table, each row represents a traced signal (A for instance). Using data expansion methods similar to [65], such as forward and backward propagation of the traced data, Table 3.2 can be expanded to hold all the other signals. Table 3.3 is an example of such an expanded trace. In this table the first 3 rows are the exact copy of the traced flops (i.e., A, B, C). The additional rows (D, E, ...) belong to the untraced signals. At every clock cycle, the value of an untraced signal can either be logically inferred using all the traced values and all the previously inferred ones, or it remains unknown. As a result the expanded history contains `0` and `1` for known values and question marks for yet-not-inferred values. When building the expanded trace history, initially all the untraced signals are marked with question marks "?" to represent their unknown status. As the data expansion algorithm operates on the given trace, some of the "`?`"s will be replaced by either "`0`" or "`1`". In an ideal situation every question mark will be replaced, however with limited signals to trace some might remain unknown.

3.1.4 SAT Formulation and Its Use for Post Silicon Debug

In this section we describe how a logic circuit can be translated to SAT clauses. We also explain how the values from the failing trace can be added to the SAT formulation

Table 3.2: Trace History

Signal ↓ Cycle →	1	2	3	4	5	6	...	n
A	0	1	0	0	1	0	...	1
B	0	0	0	1	0	0	...	0
C	1	1	0	1	0	1	...	0

Table 3.3: Expanded History

Signal ↓ Cycle →	1	2	3	4	5	6	...	n
A	0	1	0	0	1	0	...	1
B	0	0	0	1	0	0	...	0
C	1	1	0	1	0	1	...	0
D	?	?	1	0	0	?	...	?
E	?	1	?	?	?	1	...	0
⋮	⋮							

that contains the logic circuit.

Boolean relationships can be written in a format known as Conjunctive Normal Form (CNF), which describes every Boolean relation as a set of logic conditions that must hold altogether. Table 3.4 shows these conditions for basic logic gates. Here we illustrate the relationship using a sample circuit. Figure 3.2 shows a circuit containing 4 flip-flops and 2 logic gates. If we denote F_T as the value of flip-flop F at clock cycle T (at the Q terminal), the following relationships, as defined by the `AND` and `OR` gates from the circuit from figure 3.2,

$$C_{T+1} = A_T \cdot B_T$$

$$D_{T+1} = A_T + B_T$$

Table 3.4: Conjunctive Form of Basic Logic Gates

Logic	Definiton	Conjunctive Form
AND	$y = a \cdot b$	$(a+\overline{y})(b+\overline{y})(\overline{a}+\overline{b}+y) = 1$
OR	$y = a + b$	$(\overline{a}+y)(\overline{b}+y)(a+b+\overline{y}) = 1$
NOT	$y = \overline{a}$	$(\overline{a}+\overline{y})(a+y) = 1$

can be transformed to the logic conjunction (AND) of the following six clauses:

$$(A_T + \overline{C_{T+1}}) \quad \cdot \tag{3.1}$$

$$(B_T + \overline{C_{T+1}}) \quad \cdot \tag{3.2}$$

$$(\overline{A_T} + \overline{B_T} + C_{T+1}) \quad \cdot \tag{3.3}$$

$$(\overline{A_T} + D_{T+1}) \quad \cdot \tag{3.4}$$

$$(\overline{B_T} + D_{T+1}) \quad \cdot \tag{3.5}$$

$$(A_T + B_T + \overline{D_{T+1}}) \quad = 1 \tag{3.6}$$

A SAT clause is a logic disjunction (OR) of literals, where a literal is a Boolean variable (which may or may not be complemented). The first three clauses from the above SAT formulation formulate the AND gate from figure 3.2 and the last three clauses represent the OR gate. In the SAT formulation, all the clauses are ANDed together. A SAT solver will search for an assignment of the Boolean variables to determine if all the clauses can be true at the same time.

The trace bits collected during the post-silicon debug can also be added to the SAT formulation as a set of additional clauses. Table 3.1 shows two sets of assumptions for values of our flip-flops. We have transformed every single assumption for the values of

a flip-flop F at any time T into a single condition of form (F_T) or $(\overline{F_T})$ based on the flop's value. These conditions, that hold only a single variable are called unit clauses and are easy to evaluate since they simply assign their variable to a logical 0 or 1, without any necessary knowledge about the other variables. Within a SAT solver, these unit clauses will cause a Boolean constraint propagation, where the set of the remaining clauses will be reevaluated/updated, i.e., some clauses might be removed if they are satisfied as a consequence of the evaluation of the unit clause or the number of literals will be reduced because the Boolean variable assigned in the unit clause is not unknown any more.

It can be verified that the values in the left half of Table 3.1, or equivalently the conditions in the bottom left of this table, can satisfy all the of the conditions in the SAT problem. However, the right half of table 3.1 will not: knowing that $C_{T+1} = 1$ and then satisfying conditions (3.1) and (3.2), we conclude that $A_T = B_T = 1$, which according to either of conditions (3.4) or (3.5) leads to $D_{T+1} = 1$ which is a contradiction with the given data in table 3.1. In others words there are no two values of A and B that have `AND(A,B)=1` and `OR(A,B)=0`.

In the above example the SAT problem is known as "Unsatisfiable", or UNSAT for short. A set of clauses leading to a contradiction are known as a **Core of Unsatisfiability** (or the core of UNSAT). For the above example, the core of UNSAT contains the two conditions in the bottom right of table 3.1 and conditions {3.1,3.4} of the listed problem, thus a total of 4 conditions:

$$(A_T + \overline{C_{T+1}}) \cdot (\overline{A_T} + D_{T+1}) \cdot (C_{T+1}) \cdot (\overline{D_{T+1}}) \neq 1$$

Technically the SAT solver is free to choose the order in which it evaluates the clauses of a Boolean SAT problem. Consequently, different solver configuration or

decision algorithms may find a different Core of UNSAT. In order to show that the core of UNSAT is not necessarily unique, even in this small example, one can verify that the core could include conditions 3.2 and 3.5 instead of 3.1 and 3.4:

$(B_T + \overline{C_{T+1}}) \cdot (\overline{B_T} + D_{T+1}) \cdot (C_{T+1}) \cdot (\overline{D_{T+1}}) \neq 1$

The Boolean variables in the core of UNSAT capture both the flip-flops and the clock cycles that are contributing to the logic inconsistency in the failing trace and can thus provide a narrowed down list of suspects (in both space and time) for the root cause of the problem. And this can be achieved without needing a separate debug session, which is often the case when the bit-flip has been induced by a subtle electrical problem that is difficult (if not infeasible) to reproduce.

3.2 Methodology and Evaluation Platform

Figure 3.4 gives an overview of the steps in our methodology. As seen in the example from the previous section, a SAT formulation comprises two categories of clauses. First set are the clauses based on the circuit structure, that captures how the logic gates are connected to each other. The second group of clauses are produced by the failing trace that was collected. Once the two categories of clauses are merged and the final SAT formulation is created, any third-party SAT solver can be used to search for a satisfiable assignment of the Boolean variables. In the event that the problem proves to be *Unsatisfiable*, the core of UNSAT is analyzed further. In the following we will explain each of these steps in detail.

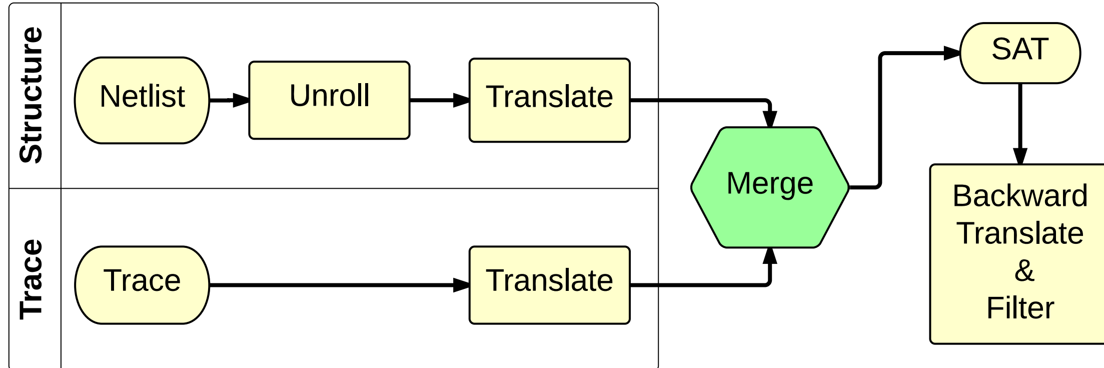


Figure 3.4: The SAT formulation consists of two sets of SAT clauses: From the circuit structure and from the failing trace.

3.2.1 Circuit Unrolling

In a digital circuit each flip-flop holds a different value in each clock cycle. Consider a flip-flop named "A", as depicted in figure 3.5. During n clock cycles, we have virtually n different flip-flops "A_i", one for each clock cycle. At any given cycle, the value on the `Q` outputs of every flip-flop feeds the combinational logic connected to it to form the next state logic, by providing a value to the `D` inputs of all flops within the same clock cycle. The `D` values of cycle T become the `Q` values of the cycle $T + 1$. As a result, to formulate a complete SAT problem from a circuit, we need to know the trace depth, n, and we must unroll the circuit to n subcircuits and connect the combinational logic inside each subcircuit layer, while the only interconnection between the subcircuits are through the sequential elements, i.e., flip-flops.

3.2.2 Translation

Circuit

Table 3.4 shows the CNF representation of the basic logic gates. These are known as Tseytin transformations [89]. This table is used to translate the basic logic elements in a circuit into Boolean SAT clauses. Table 3.5 shows the same type of translation rule for larger logic gates and flip-flops. As it can be seen for relatively larger logic gates, the SAT representation requires more clauses (in particular for `XOR` gates) and more literals per clause. Note, we can decompose a larger logic block into smaller ones and then write the translation for the smaller unit, which comes at the expense of having more Boolean variables for the intermediate signals in between the smaller units. For example, we may replace a 4-input `AND` with three smaller 2-input `AND` gates.

Trace

For the trace subproblem we have to extract the 0/1 values for every traced flip-flop from the trace memory and based on the value we introduce $(F_T) = 1$ or $(\overline{F_T}) = 1$ for flip-flop F at clock cycle T. As it is the case for the circuit SAT subproblem, each

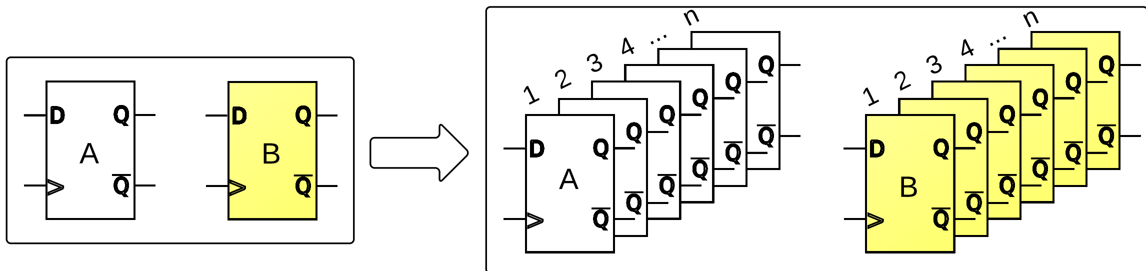


Figure 3.5: All the flip-flops are unrolled for as many times as the trace depth.

Table 3.5: Conjunctive Form of Larger Logic Gates and D-type Flip-Flop

Logic	Definition	Conjunctive Form
D/Q FF	$Q_{T+1} = D_T$	$(\overline{Q_{T+1}} + D_T)(Q_{T+1} + \overline{D_T}) = 1$
OR3	$y = a + b + c$	$(y + \overline{a})(y + \overline{b})(y + \overline{c})$ $(\overline{y} + a + b + c) = 1$
OR4	$y = a + b + c + d$	$(y + \overline{a})(y + \overline{b})(y + \overline{c})(y + \overline{d})$ $(\overline{y} + a + b + c + d) = 1$
AND3	$y = a \cdot b \cdot c$	$(\overline{y} + a)(\overline{y} + b)(\overline{y} + c)$ $(y + \overline{a} + \overline{b} + \overline{c}) = 1$
AND4	$y = a \cdot b \cdot c \cdot d$	$(\overline{y} + a)(\overline{y} + b)(\overline{y} + c)(\overline{y} + d)$ $(y + \overline{a} + \overline{b} + \overline{c} + \overline{d}) = 1$
NOR3	$y = \overline{a + b + c}$	$(y + a)(y + b)(y + c)$ $(\overline{y} + \overline{a} + \overline{b} + \overline{c}) = 1$
NAND3	$y = \overline{a \cdot b \cdot c}$	$(\overline{y} + \overline{a})(\overline{y} + \overline{b})(\overline{y} + \overline{c})$ $(y + a + b + c) = 1$
XOR	$y = a \oplus b$	$(\overline{y} + a + b)(y + \overline{a} + b)$ $(y + a + \overline{b})(\overline{y} + \overline{a} + \overline{b}) = 1$
XOR3	$y = a \oplus b \oplus c$	$(\overline{y} + a + b + c)(y + \overline{a} + b + c)$ $(y + a + \overline{b} + c)(y + a + b + \overline{c})$ $(y + \overline{a} + \overline{b} + \overline{c})(\overline{y} + a + \overline{b} + \overline{c})$ $(\overline{y} + \overline{a} + b + \overline{c})(\overline{y} + \overline{a} + \overline{b} + c) = 1$

flip-flop has its own Boolean variable for each clock cycle.

A major difference between the two SAT subproblems, apart from the origin of the conditions, is the fact that the latter consists of only unit clauses whereas the former category contains larger clauses depending on the size of logic gates.

3.2.3 Merging and Running the SAT Solver

Merging the two subproblems is trivial. There are certain rules that have to be followed to ensure the problem is compatible with standard off-the-shelf SAT solvers. Here we formulate the SAT problem in simplified DIMACS format [90], in which all

the variables are replaced by numbers. The specific implementation details are not discussed here.

To solve the SAT problem we use the and open-source solvers *"PicoSAT"* [91] and *"miniSAT"* [92]. Apart from user friendliness and good performance, the fact that PicoSAT generates the core of UNSAT has been a major reason for us to consider this solver.

Finally, the list of variables inside the core of UNSAT are extracted and used for further analysis. Note that if the problem is satisfiable, this method has no means to identifying any design bug. A satisfiable assignment means that the traced signals and the circuit are consistent, which shows that the collected trace is not sufficient to find the logical inconsistency.

3.2.4 Backward Translation and Filtering

The core of UNSAT, contains the list of conditions which were incompatible. The next step for us is to backward-translate the list of variables in this list to the original netlist domains. There is a one-to-one mapping between each Boolean variable in SAT problem and a signal in the unrolled circuit. As a result each UNSAT variable points to a net, either a logic gate or a flip-flop, at a specific clock cycle. This 2-dimensional list of net and time pairs can provide a set of candidates of the root cause of the inconsistency. We can further filter this list to hold only the flip-flops.

3.2.5 Evaluation Platform

In order to evaluate the steps discussed in this section, we needed to design a (virtual) evaluation platform in which it is simple to inject bit-flips into any of the flip-flops

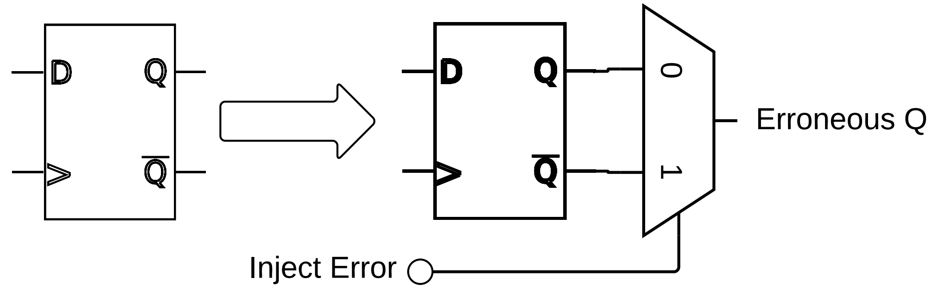


Figure 3.6: Flip-flops are replaced by error-injectable flip-flops.

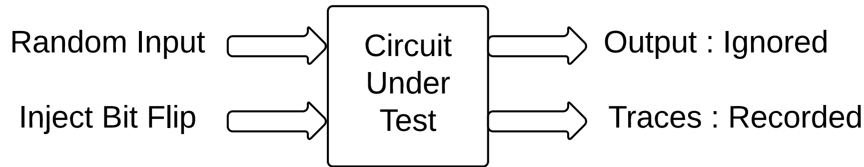


Figure 3.7: Bit-flips are injected to the circuit-under-test.

in a test circuit. We parse the input netlist and replace the flip-flops with the blocks shown in figure 3.6. In order to inject a bit-flip into a flip-flop we have to assert the "Inject Error" bit of that particular flip-flop to a logic `1` at a specific time, for a duration of one clock cycle.

Then we run multiple simulations of our test circuits, applying random inputs and at the same time collecting traces. At some arbitrary point in time we inject a single bit-flip into one of the flip-flops. We also have to choose a number of flip-flops as trace signals, according to our predefined trace budget. In this chapter we assume that the traced flops are selected randomly. When the simulation is over we send the collected trace and the circuit to the translation engine to generate the SAT formulation. The SAT solver is called to analyze the problem and, if the problem was unsatisfiable the core of UNSAT is sent to backward translation engine and the list of candidate flip-flops are extracted.

3.3 Experimental Results

We ran experiments on different benchmark circuits including ISCAS '89 `s9234, s38584, s38417` [93] and ITC '99 `b21`, [94]. We used commercial simulator and computer-generated SystemVerilog hardware description language. All the developed programs are in C/C++, with some automated scripts written in Bash under Linux.

In an experiment we inject a single bit-flip into every flip-flop in the circuit in separate simulations. This means that in each simulation run, only one of the flip-flops is affected by the bit-flip. As a result, for a circuit with n flip-flops an experiment consists of n different simulations. We randomly chose t out of n flip-flops as the traced signals. Which signals are the most suitable to be traced for bit-flip diagnosis is a standalone problem that is discussed in detail in the next chapter. Here, we focus on showing the feasibility of using the core of UNSAT for narrowing down where and when the bit-flips have occurred. To keep the experiment self-consistent these t traced signals are kept the same within a single experiment; the same applies to the error injection time and the trace collection window. In all these experiments the trace collection windows, also known as trace depth, is 100 cycles and the error is injected at cycle 50.

In each simulation the trace is recorded and undergoes the steps shown in figure 3.4. Next, the n SAT problems are generated and passed to the solver. Finally the UNSAT problems are counted and kept for further processing. Figure 3.8 summarizes the percentage of the problems that proved to be UNSAT as a result of increasing the trace width. The horizontal axis shows the ratio of the signals that are recorded in each experiment. As it is expected, the number of UNSAT problems grows as the number of traced signals increases. The reason is that the solver is provided with

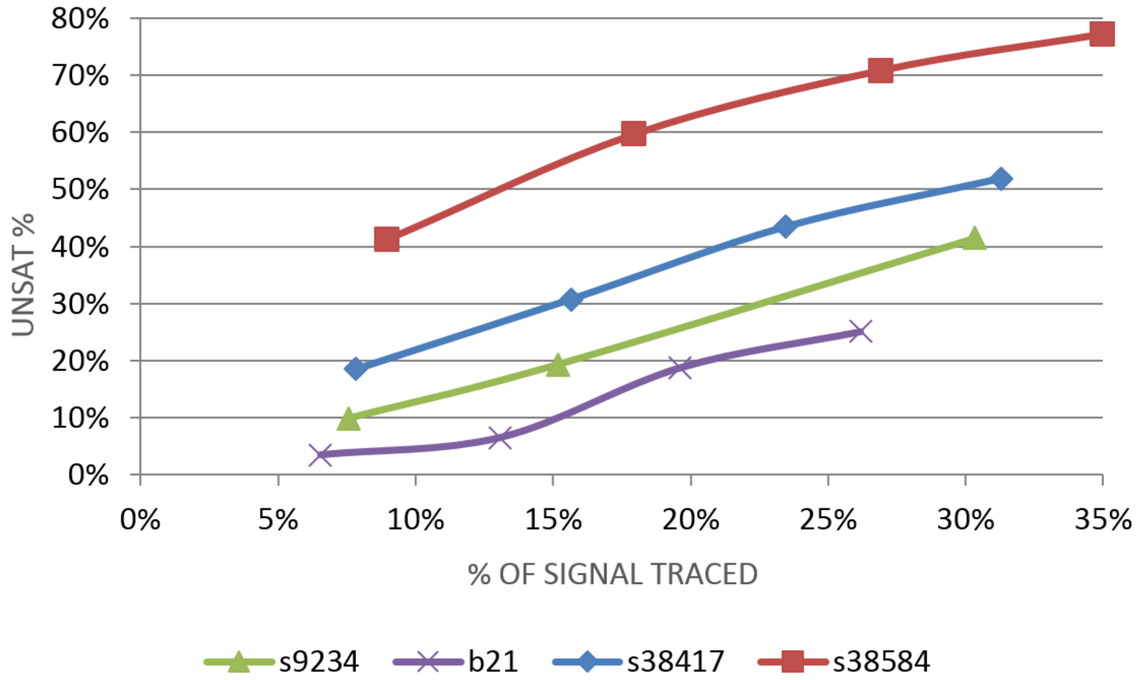


Figure 3.8: Ratio of UNSAT problems vs the ratio of signals being traced.

more input data to find the inconsistency. It is shown that different circuits behave differently in this experiment, however the monotonic increase in the UNSAT ratio is consistently observed.

When a problem proves to be UNSAT, its core of UNSAT is then further processed. The variable names are backward-translated to nets and times of the reported inconsistency. The analyzer lists the flip-flops involved in the core and their corresponding time-tag, which is defined as the *candidate group*. Bug localization requires the ability to shrink the candidate group both in time domain as well as flip-flop count. Figure 3.9 shows that the majority of these candidates are within a distance of 5 cycles from the cycle that the error was injected. This chart is for circuit `s38417` with a trace width of 128 bits (out of 1636 flip-flops).

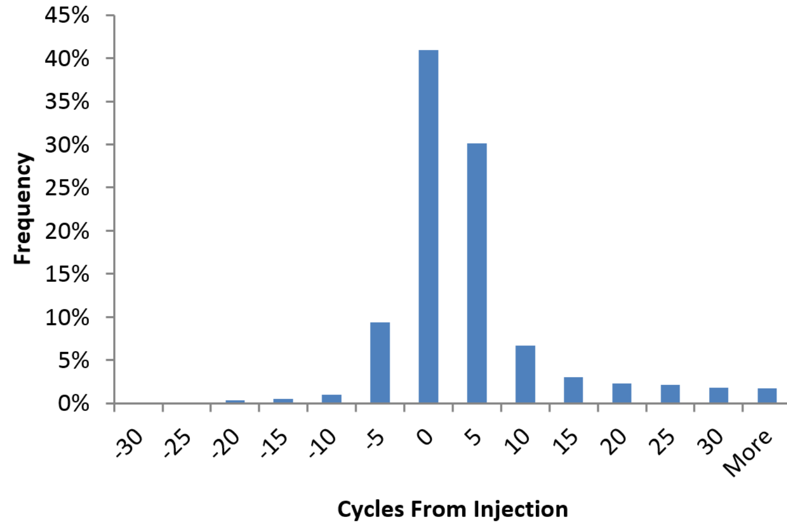


Figure 3.9: Distribution of the candidate group in time with respect to the bit-flip injection at cycle 0.

Figure 3.10 shows the histogram of the number of suspect flip-flops in the candidate group for s38417. This figure shows that the increase in trace width, apart from the increased chance of error detection, shrinks the size of candidate group. With trace width of 128 and 256, respectively, the size of the candidate group is less than 30 and 20, for approximately 85% of the experiments that have produced a failing trace where a logic inconsistency was found.

Another result worth reporting is the runtime of the SAT solver. From the practical standpoint, it is necessary to understand which kind of experiments influence the runtime. Since the overall runtime is dominated by the SAT solver step, we will only report the time required by PicoSat to return a satisfiable assignment (when no logic inconsistency was found) or the core of UNSAT (when a logic inconsistency was identified). It is interesting to observe that generally the unsatisfiable problems were proven faster than the satisfiable ones. Figure 3.11 summarizes these measurements.

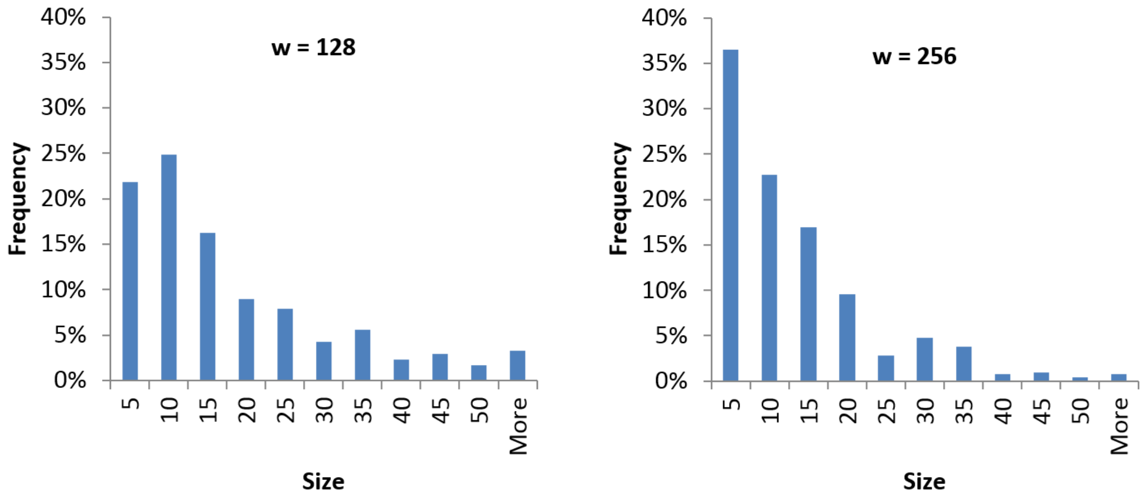


Figure 3.10: Distribution of the number of flip-flops in the candidate group for two trace widths (left:128 and right: 256).

Although the runtime is dependent, in part, on the internals of the SAT solver, we believe that the solver is spending more time to find a difficult solution to satisfy assignments (for traces were a bit-flip was injected), rather than more rapidly identifying a logic contradiction, as a result of a signal inconsistency (at which point the search engine within the SAT solver stops).

The above experiments were run remotely on a server farm with (mostly) AMD Opteron 2.2 GHz processors. It was observed that the majority of instances used between 600 MB and 1.2 GB of memory. When increasing the trace depth from 100 cycles to 200 and then to 1000 cycles, the memory usage increased to around 2 GB and 8 GB respectively. The average UNSAT detection time also increased from 5 seconds to 46 and 564 seconds respectively. It is worth noting that after analyzing the core of UNSAT we have observed that the size of the candidate group varies insignificantly when varying the trace depth. To summarize, when increasing the

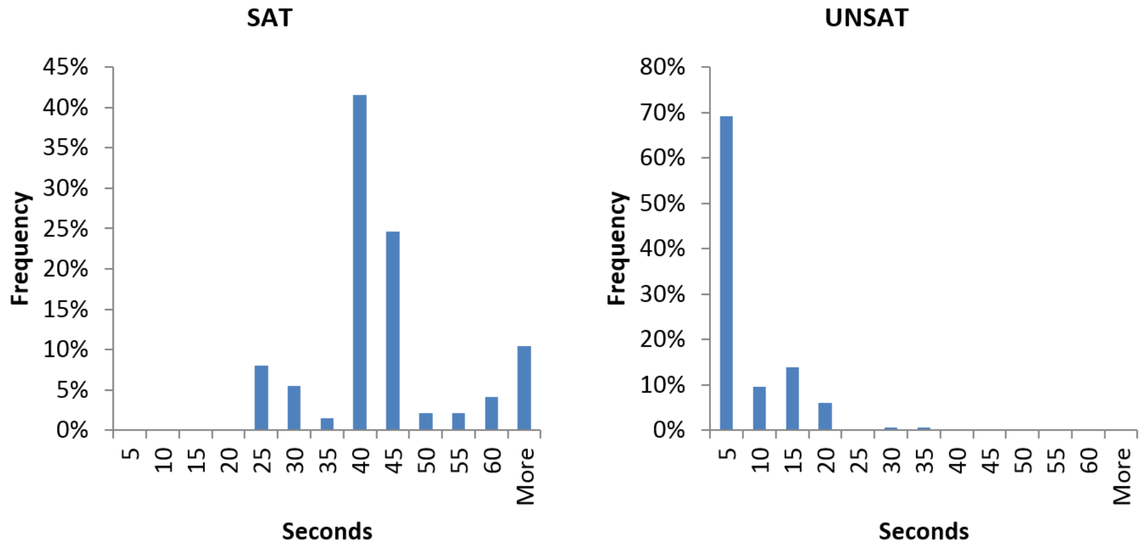


Figure 3.11: Histogram for the runtime of the SAT solver, in seconds, for SAT and UNSAT problems (left: SAT, right: UNSAT).

trace depth, a SAT instance that is 10 times larger leads to approx 100 times more runtime and approx 10 times more memory usage.

3.3.1 The Non-Uniqueness of the Core of UNSAT

As mentioned earlier the core of UNSAT is not necessarily unique, since the order of processing the SAT clauses is not uniquely determined for each SAT run. We conducted a few experiments to find and analyze different cores of UNSAT. By manipulating the order of clauses, through changing a random seed provided to PicoSAT [91], we were able to extract different cores. First, for a smaller circuit `s5378` with 179 flip-flops, we attempted to extract 100 different cores for 90 different UNSAT instances after a single bit-flip injection (a total of 9,000 PicoSAT runs). For each instance, all the 100 cores of UNSAT were exactly the same with an average size of 5

flip-flops in the corresponding list of suspects.

For `s38417`, for a number of instances where the size of original core of UNSAT had variables derived from over 30 flip-flops, 100 cores were identical. For `s35932`, however, a variation in size of the cores of UNSAT can be observed. First, for relatively smaller cores, where the core of UNSAT maps to a suspect list of less than 30 flip-flops, no variation in the size of the core of UNSAT was observed. However, for instances that had a list of suspects of around 50 flip-flops, different cores were extracted with the corresponding suspect list ranging from 43 to 189 flip-flops. We suspect that for several of the relatively larger suspect lists for `s35932`, there might be a smaller counterpart which can potentially be extracted with a different SAT solver or with a different configuration of the same SAT solver. Note that the SAT solver used for this experiment (picoSAT [91]) accepts a random seed parameter which affects the order of its internal decisions, thus potentially different cores of UNSAT would be generated.

3.4 Summary

In this chapter the following topics were discussed :

- Systematic approaches are necessary to be developed for post-silicon validation. In this contribution we focused on automatic detection of bit-flips in a digital circuit. We rely on an on-chip trace buffer that records several signals as the circuit is in operation.

- After a failure, the recorded trace bits are extracted and post-analyzed to produce a list of suspect flip-flops where the bit-flip has probably occurred.

- The methodology relies on Boolean SAT solvers for detection of the bit-flip and the Core of Unsatisfiability generated by such solver is used to produce the list of suspects.
- A netlist first needs to be translated to a Boolean SAT problem using developed computer programs before a solver attempts to solve it. Later, another piece of developed software post-processes the Core of UNSAT to extract list of suspects.
- Results from the conducted experiments prove the effectiveness of this method. Thus, in the next chapter we focus on selecting a set of trace-signals that maximizes bit-flip detection.

Chapter 4

Trace Signal Selection

4.1 Background

In chapter 2 it was explained that on-chip tracing has become a common method for post-silicon validation of digital integrated circuits. In a nutshell, an on-chip memory can be used to record the internal nets of a circuit and therefore keep a sequence of events. The recorded content can be used for debug purposes when necessary and provide insight into the problem at hand. However, due to the incurred cost only a small fraction of all the signals within a design can be recorded. As a result, the number of traced signals is a budget that should be spent wisely. A designer may use his inner knowledge of the system to select key signals. Alternatively one may use algorithms that automatically chooses signals with a certain objective, since there may be non-obvious logical relations among a group of signals that are not easily seen by the designer. It was also discussed that there are several other works in which trace signals are selected, however are either tuned towards a different objective or based on a different set of assumptions from the work proposed in this thesis.

As discussed in chapter 3, a systematic methodology has already been established to detect bit-flips within a digital IC. It was shown through experiments that even a random set of traced signals may detect a bit-flip and to some extent provide a list of suspect flip-flops. This detection method provides insight during root-cause analysis of the failing trace. As highlighted in figure 4.1, in this chapter we will explain a systematic method for selecting traced signals. It is assumed that during the design time the designer needs a tool that processes the netlist and automatically chooses the "best" signal to be selected.

Our objective is to increase the likelihood of detecting more bit-flips. As a result the main contribution of our work is to define new metrics that can guide trace signal selection algorithms that will assist with bit-flip detection and localization. The key insight lies in the fact that trace data collected during a debug experiment should imply opposite values in the flip-flop where the bit-flip has occurred.

This chapter is organized as follows. First, we will provide a few examples on bit-flip detection and localization, followed by a new trace signals selection algorithm that prioritizes bit-flip detectability. Finally, our results show the effectiveness of our algorithms for detecting and localizing bit-flips in a collected trace.

4.2 Motivational Example

4.2.1 Restoration Ratio vs Bit-flip Detectability

It has been shown how solving a SAT instance based on a circuit netlist and a binary trace can help with the detection and localization of bit-flips. The effectiveness of this approach relies on the size and quality of the collected trace. Since the number

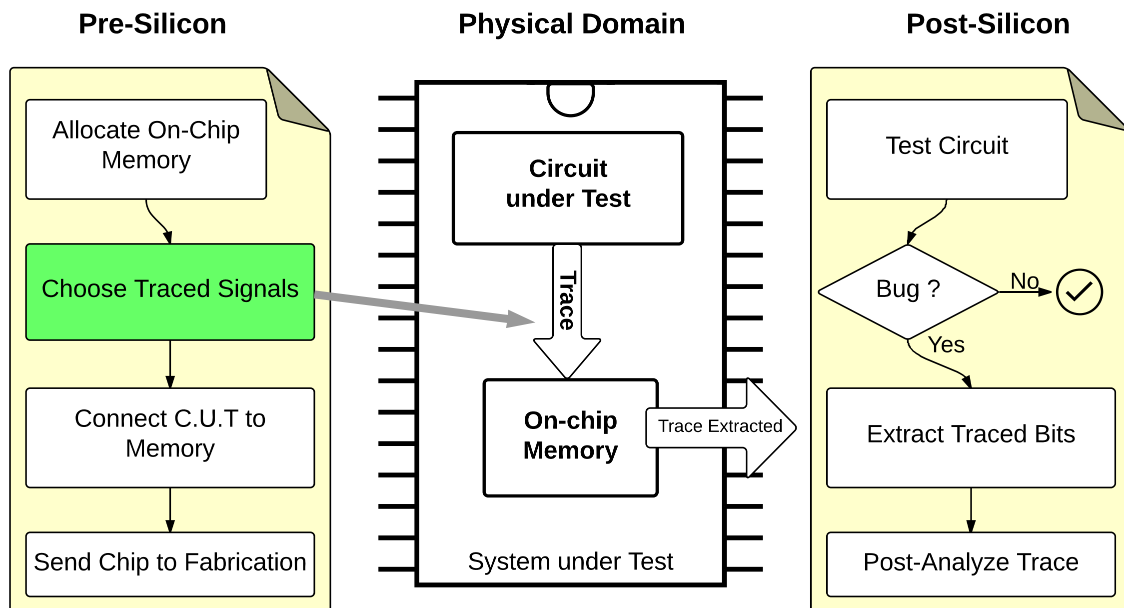


Figure 4.1: Pre-silicon and post-silicon tasks for trace-based debugging. Choosing which signals to trace is covered in this chapter.

of signals that can be traced is limited, a natural question that arises is which signals to trace. As outlined in the introductory chapter, there have been numerous studies over the past decade on trace signals selection. In this section we illustrate why a high restoration ratio used by the known art, i.e., the ability to restore the logic values for many signals in the design across multiple clock cycles, is a necessary but not a sufficient condition for bit-flip detection.

Figure 4.2(a) shows a 3-bit shift register. Let us assume that only flip-flop `A` is *traced*, which means that its value is recorded in an on-chip memory at every clock cycle during the acquisition window. Figure 4.2(b) shows a sample history of flip-flop `A` for a window of 6 clock cycles. We can infer the values of flip-flops `B` and `C` starting from cycles 1 and 2, respectively. In this scenario the values of `B` and `C` are partially *restored*, with restoration ratio of $\frac{4+5+6}{6}$ for this example (figure 4.2c). This is, of course, under the assumption that no bit-flip occurred and the restored data can be used to reason whether the observed behavior matches the intended behavior (hence, the usage for functional bugs). If, however, flip-flop `B` is affected, for example, by a bit-flip at cycle 2 (as in figure 4.2(d)), this incorrect value is passed also to flip-flop `C` in the physical device, which would be inconsistent with what is restored only from flip-flop `A`.

For the 3-bit shift register example from figure 4.2, we would be able to detect any bit-flips in flip-flops `B` and `C` if, in addition to flip-flop `A`, we trace also flip-flop `C`, as shown in figure 4.2(e). More generally, tracing the first and the last flip-flop can identify any single bit-flip within a shift register structure (except in the first flip-flop). As shown in figure 4.2(f) forward propagation of `A` at cycle 1 together with backward propagation of `C` at cycle 3 provide proof that an inconsistency exists. This

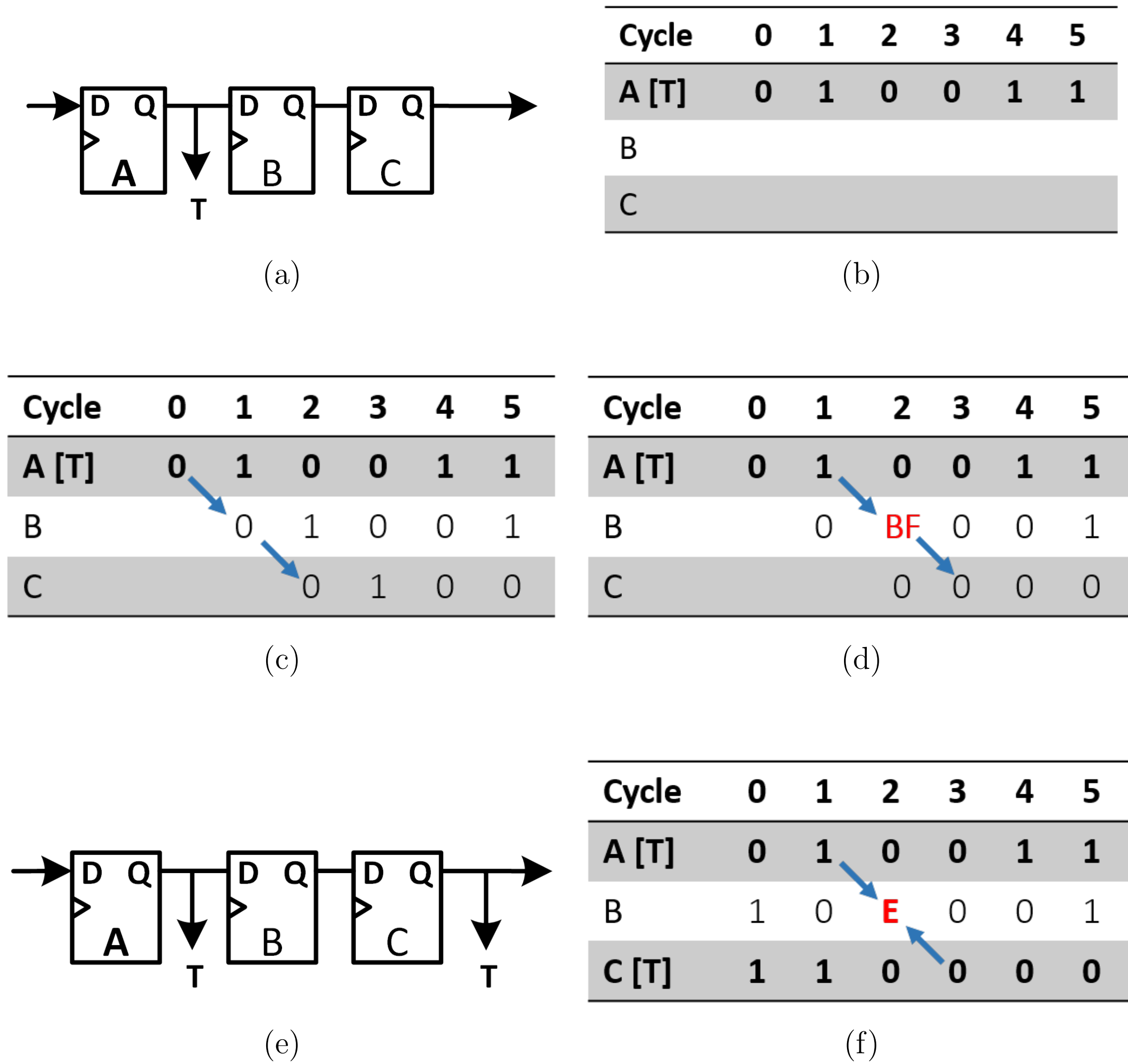
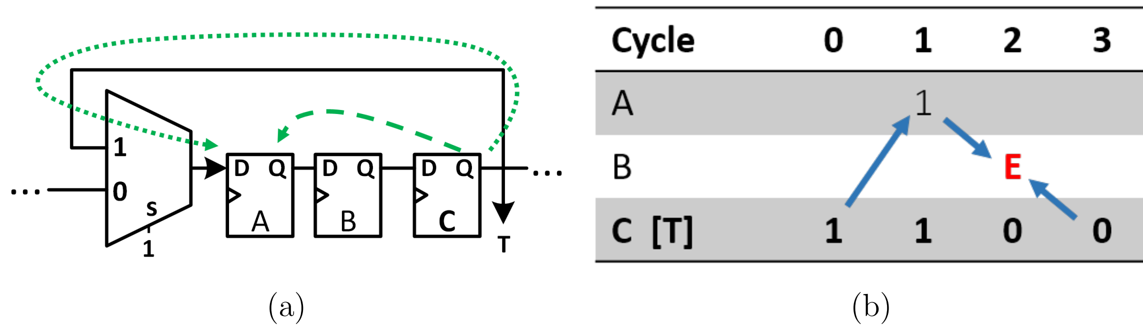


Figure 4.2: Example circuit with trace table. Traced signals are marked with [T]. (a) Shift register with 3 flip-flops and flip-flop A is traced. (b) Trace bits recorded in the on-chip memory. (c) Same as (b), but after data restoration. (d) Trace table when bit-flip occurs in flip-flop B. (e) Flops A and C are traced. (f) Bit-flip error is detected and marked with E.



(a) (b)

Figure 4.3: (a) Circular shift register. Forward and backward propagations are marked. (b) Error detected using multiple bits of a single trace signal.

proof becomes available when we take advantage of multiple sources of information to restore `B` to different (opposite) logic values.

For the example from figure 4.2 we needed to trace a single flip-flop in order to restore all the bits in the shift register, and we needed to trace two flip-flops (first and last) in order to enable bit-flip detectability. Intuitively, one would expect more signals to be traced in order restore the flip-flop where the bit-flip occurred to opposite values. Nonetheless, this is not necessarily the case, as shown in figure 4.3, where we have a circular shift register. We assume the mux is in rotate mode by applying logic 1 to the select signal. In this circuit only flip-flop `C` is traced for which the recorded vector is $[1, 1, 0, 0]$. As we reconstruct the circuit's state using multiple forward and backward propagations, we end up with the inconsistent value for `B` at cycle 2. The steps are shown in figure 4.3(b).

4.2.2 Key Insight

An important point articulated through the above examples is that restoration is a necessary but not sufficient for bit-flip detection. While restoration ratio was proposed

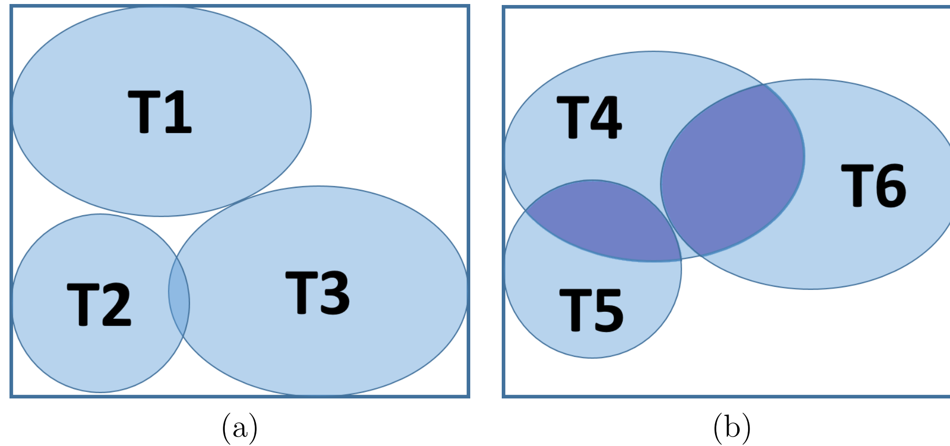


Figure 4.4: (a) T1-3 partially restore circuit's state. (b) Signals in the intersection of T4-6 can be implied twice, hence a possibility for error detection. We assume a second restoration originates and propagates from a second port/path.

initially in [65] as a quantifiable (and objective) metric to assess the suitability of a set of trace signals for post-silicon validation, it is by no means a one-fit-all metric. A similar point was made recently in [77], where the objective was to improve functional coverage (rather than bit-flip detection). The key insight for bit-flip detection is that data must be restored using **two witnesses**: one witness implies that a flip-flop must hold a `0`, while the other one forces the same flip-flop to `1` in the same clock cycle. In terms of using a SAT engine for bit-flip detection, the aim is to use unit clauses (derived from collected trace) that will increase the probability of producing an unsatisfiable outcome, whenever a bit-flip occurs.

Figure 4.4 illustrates the restoring region of two different sets of trace signals with similar restoration areas. Each trace signal, Ti, is assumed to restore a region of the circuit shown by oval shapes. Traces T1-T3 have relatively smaller overlap and together they cover an area A_{123} of the left rectangle. The coverage area of each individual trace in figure 4.4(b) is intentionally set to be the same of their counterpart in figure 4.4(a). Since traces T4-T6 have higher overlap area compared

to traces T1-T3, their overall covered region, A_{456}, is smaller than A_{123}. Nevertheless, the overlapped regions are larger and represent that some flip-flops can potentially be restored from both ends. Only the flip-flops in these overlapped regions have the potential for bit-flip detection.

In figure 4.2(e) the D port of flip-flop `B` is restored through our knowledge of flip-flop `A` in cycle 1, or `A[1]`. We call this effect *D-restoration* of flip-flop `B`. Similarly Q port of flip-flop `B` is restored by flip-flop `C[3]`. We detected the bit-flip due to **simultaneous D and Q-restoration** of `B`. In this example the two witnesses are `A[1]` and `C[3]`. In a shift register structure, when one flip-flop is traced, all the following flip-flops on its right become D-restorable. Similarly all the previous flip-flops on the left benefit from being Q-restorable. It is also worth mentioning that, since trace data is collected over several clock cycles, different bits of the same signal that is traced can play the role of any of the two required witnesses (see figure 4.3).

Due to the importance of having two witnesses for each flip-flop that is susceptible to bit-flips, our algorithm for trace signals selection will try to maximize the chance of restoring flip-flops on **both D and Q ports to opposite logic values**, in order to increase the likelihood of bit-flip detection. This goal is conceptually different from all the known works on trace signals selection and it provides the objective of our new algorithm described in the next section.

4.3 New Trace Signals Selection Algorithm

Before discussing the new algorithm, we introduce the definitions needed to formalize bit-flip detectability, and we present the basic rules for processing the netlist in order to compute the bit-flip detectability for each flip-flop.

4.3.1 Definitions

We first define the *zero* and *one-restorability* of a signal, as well as the *certainty* and the *average value*. All these definitions are needed to define *bit-flip detectability*. When dealing with signals that may or may not be restored, there are 3 possible outcomes for each signal. A signal may be restored to 0 or 1 or may stay unknown. The following definitions help us formulate probabilistic estimates for each of these 3 outcomes.

Zero- and One-Restorability

We define the zero-restorability ($R0$) of a signal as the probability of restoring the respective signal to logic zero. Likewise, we define the one-restorability ($R1$) of a signal as the probability of restoring the respective signal to logic one. Note that both $0 \leq R0 \leq 1$ and $0 \leq R1 \leq 1$.

Certainty of a Signal

We define *Certainty* (C) of a signal as the probability of recovering a signal form the recorded trace. The certainty C equals by definition $R0 + R1$. Hence, hence the signal is recovered with probability $R1+R0$, and it remains unknown with probability $(1 - R0 - R1)$. Since $0 \leq C \leq 1$, a certainty of zero corresponds to a signal which is not recoverable, and a certainty of a **traced** signal is 1 because its logic value is always determined.

The Certainty values of all the untraced signals are initially assumed to be zero. Starting from the traced flip-flops, the Certainty propagates through the circuit. This propagation procedure has some rules that will be explained in the following sections.

Average Value of a Signal

For a signal that is at least partially recoverable, ($C > 0$), the *Average Value*, (V), is defined as the expected value of that signal. When a signal is more likely to be logic `1` than logic `0`, the average gets close to one and vice versa. V is defined as $\frac{1\times R1+0\times R0}{R0+R1} = \frac{R1}{R0+R1} = \frac{R1}{C}$. Together with the Certainty, V also propagates through the logic circuit. For simplicity we may use the term "propagation of *Certainty*", whenever we refer to the propagation of the (C, V) tuple. For the rest of this text and the pseudo-codes that will follow, the notation CV is used to refer to this tuple and CV_x denotes the certainty and value of net x.

Certainty and Value are independent as illustrated in the unit square of Figure 4.5(a). The one-restorability is the intersection of probability of recovering C and the probability of signal being `1`. This is the area labeled as $R1$ in the figure. Likewise, the zero-restorability is shown as $R0$. The area labeled with a question mark represents the probability of a signal staying unknown. As discussed above, there is a one-to-one mapping from any (C, V) tuple to an $(R1, R0)$ tuple and vice versa. Since $C = R1 + R0$ and $V = \frac{R1}{C}$ we have $R1 = C \times V$ and $R0 = C \times (1 - V)$. In our algorithm we will frequently refer to all the above defined concepts and the following example helps clarify them.

- Example I: Signal `A` is recoverable half the times and there is $\frac{1}{4}$ chance that when it is recovered it becomes `1`. In Figure 4.5(b) we have:

  $C_A = \frac{1}{2}, V_A = \frac{1}{4} \Rightarrow R1_A = \frac{1}{8}, R0_A = \frac{3}{8}$

- Example II: Signal `B` is non-recoverable since there is no trace in its vicinity to restore it.

  $C_B = 0, V_A = \text{Undefined} \quad \Rightarrow R1_B = 0, R0_B = 0$

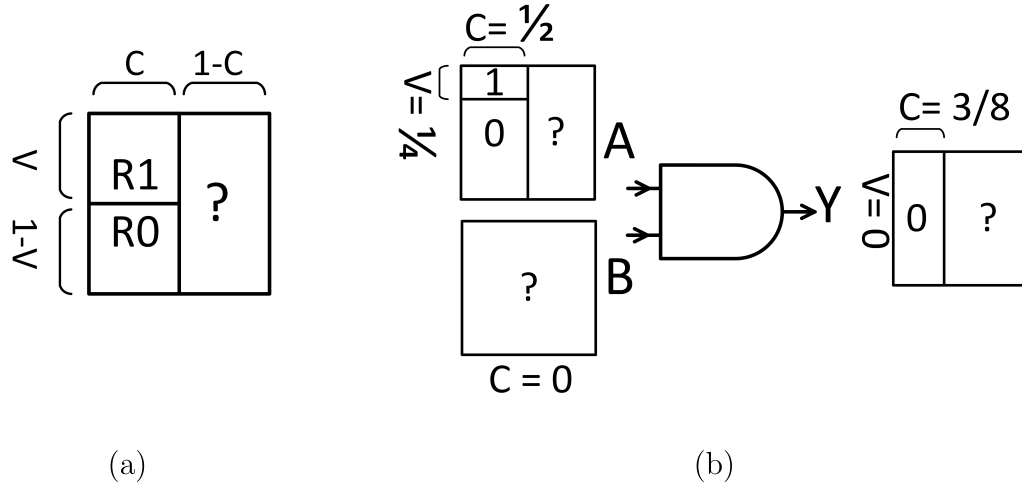


Figure 4.5: (a) Representation of Certainty (C) and Value (V) as probabilities. (b) Examples I-III showing how `Y` is only partially zero-restorable.

- Example III: Signals `A` and `B` are inputs to an `AND` gate to form output `Y`. Every time `A` is `0`, regardless of what `B` is, `Y` is also recovered to `0`. Also, since `B` is not restorable to `1`, `Y` is also not restorable to `1`. Using the propagation rules that are elaborated in the next sub-section we have:

  $R1_Y = R1_A \cap R1_B = 0, R0_Y = R0_A \cup R0_B = \frac{3}{8}$

  $\Rightarrow V_Y = 0, C_Y = \frac{3}{8}$

  We refer to `Y` as *partially* zero-restorable ($R0_Y$ is non-zero) and it is not one-restorable ($R1_Y$ is zero).

A critical concept in our work is bit-flip detectability, which is defined next for flip-flops only.

Bit-Flip Detectability

A flip-flop is bit-flip detectable if it is partially restorable to `1` on one of its ports (D or Q) and partially restorable to `0` on the other port. This is achievable only by explicitly defining the restorabilities to both ports D and Q, and computing and storing them separately. **This is a key difference** between what is needed for functional debugging (restore signal values in order to increase the overall observability of the design) and what is needed to detect bit-flips (the ability to imply the value of a flip-flop on **both** of its D and Q ports to opposite logic values). Assuming Rx_y is x-Restorability of port y, our formula for bit-flip detectability (BFD) of a flip-flop is defined as:

$$\boxed{BFD = R1_Q \times R0_D + R0_Q \times R1_D} \tag{4.1}$$

4.3.2 Certainty Propagation Rules

In this sub-section we discuss the certainty propagation rules, assuming a set of flip-flops have been selected as trace signals. These propagation rules are important to obtain the zero and one-restorability for all the D and Q ports for every flip-flop in the design, which would quantify the bit-flip detectability (using the above-defined formula). These propagation rules will be used in the inner loop of the heuristic for trace signals selection discussed in the next sub-section.

Assigned to every logic element in a circuit is a data structure that stores the certainty received from the adjacent elements. The dashed squares in Figures 4.6(a)-(c) illustrate these storage units. All the combinational logic elements start with an initial condition of $C = 0$, $V = 0$. Subsequently, certainty propagation originates

from the flip-flops that are marked as "traced". In our current implementation, by default and unless more information is provided, we assume that for a traced flip-flop (i.e., $C = 1$) the probability of it being 0 and 1 are equal (i.e., $V = \frac{1}{2}$). The user of our algorithm can provide a different initial value for V, depending on his inner knowledge of the design (e.g, values derived from extensive simulations). After the traced flip-flops are initialized to $C = 1, V = \frac{1}{2}$, each propagation request, issued by any logic element is stored in a queue structure. These requests are then processed one-at-a-time, until there are no more elements in the queue (note, each request may in turn generate new requests to be added to the queue). It is also important to note that during certainty propagation, logic elements frequently receive new certainty values from their adjacent elements. Therefore, a new incoming certainty is processed only if it is *improving* the C value that may have been previously received, otherwise it is ignored. In order to avoid inflated certainty values in circuits with sequential feedback loops, propagation is stopped once it traverses one full loop. Next, we elaborate the most relevant concepts for the propagation of the certainty and value tuple CV through the circuit (recall we refer to this process as certainty propagation).

Propagation Direction

Because we need to compute *both* the D and the Q restorability of every flip-flop, for each logic element certainty propagates in *both* directions (from input to output and vice versa).

Forward propagation ($\overrightarrow{fwd}$) stems from the output of a flip-flop and moves forward to the D port of the flip-flops from its output logic cone. Some of these flip-flops will become D-restorable (if their certainty value is non-zero). Similarly the non-zero

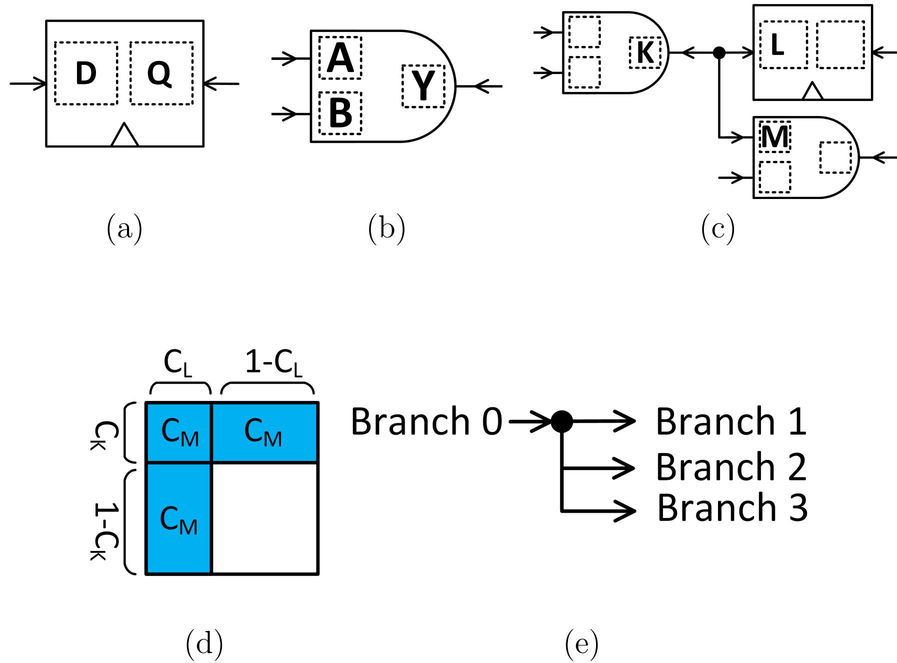


Figure 4.6: (a) Flip-flop receives certainty from D and Q. Dashed squares indicate storage that keeps the highest certainty received so far. (b) 2-input logic gate receives 3 values. (c) Gate `K` has a fan-out of 2. Each pair collectively restore the third member. (d) Highlighted area is certainty region of M. (e) Branches of multi-fan-out points are numbered, where the **stem** is labeled **Branch 0**, and the n output branches are labeled Branch 1 to n (refer to Algorithm 4).

certainty of a traced flip-flop travels **backwards** ($\overleftarrow{bkwd}$) until it reaches the Q port of the flip-flops from its input logic cone, thus making some of them Q-restorable. As a result of this distinction, we *cannot assign a single certainty* value to a net. We identify the incoming certainty at any input or output port. Inside the data structure assigned to a gate, flip-flop or a multiple-fan-out point, the received certainty is saved for later use. Outgoing certainties can be computed on-the-fly based on the stored values that have been received. Certainty propagates only in one direction at a time, with two exceptions that will be explained later for logic gates and multiple-fan-out points.

Algorithm 1 Propagation Queue

```
    Data: Certainty Propagation Queue: PropQ
 1: function PROPAGATE-ALL
 2:     while PropQ ≠ empty do
 3:         Pop front item of (PropQ) to REQ
 4:         if receiver of REQ is a flip-flop then
 5:             RECEIVE-FLIP-FLOP(REQ)
 6:         else if receiver of REQ is a gate then
 7:             RECEIVE-GATE(REQ)
 8:         else                                          ▷ Or a Fan-out point
 9:             RECEIVE-MULTI-FAN-OUT(REQ)
```

Propagations Parameters

During the propagation process, circuit elements (gates, flip-flops and fan-out points) send propagation requests to the adjacent elements. Every request, regardless of its sender or destination, contains some information that will be explained next. Note, each request may potentially generate one or more new requests. Each request *REQ* is stored in a queue data structure *PropQ* and is later processed in a first-in-first-out fashion. Algorithm 1 shows how the queue of requests is processed one request at a time until it becomes empty. Note that the elements of a circuit fall into 3 main categories (flip-flops, logic gates and fan-out branches) and each type is processed separately. This process has many steps and is broken into several distinct pseudo-codes for simpler explanation.

The content of each request is illustrated in Figure 4.7(a). Figure 4.7(b) shows an overview of the propagation process. Requests are enqueued into a queue structure and are dispatched to one of the 3 subroutines, depending on which circuit element is at the receiving end of a propagation (flip-flop, gate, branch). The very first request to start the process is generated by Algorithm 7 which selects the traced flip-flops.

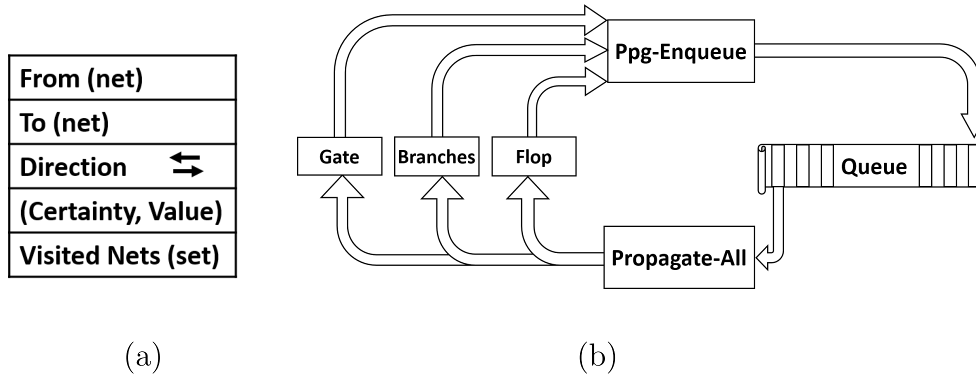


Figure 4.7: (a) The data structure of every propagation request. (b) An overview of propagation and process of the requests, using a queue.

A certainty request propagates `from` a net `to` another net, in a specific `direction` (either forward or backward). As shown in Figure 4.7(a), the request also carries a list of the nets that have already been `visited` during the previous propagation requests (starting from traced flip-flop). Each request inherits this list from its parent request and adds the next one to the list. The list helps later to detect and avoid propagation loops if the circuit has a sequential loop. If a propagation request notices that the destination net has already been visited (because the destination net is mentioned in the `visited list`), it prevents the request from generating more propagations. This mechanism must be designed in a way that certainty propagation for circuit loops: (1) is not prevented and (2) does not cause an infinite loop. On the contrary, certainty propagation for loops is necessary and must be calculated for **only one complete** loop. An example of such loop is pictured in Figure 4.10 and the details for the loop-mechanism are discussed at the end of this sub-section.

Rule 1 for Flip-Flops

A flip-flop stores two (C, V) tuples, one for each of its D and Q ports, as shown in Figure 4.6(a). The incoming certainty CV_{in} arrives at a *port*, which can be either D or Q for flip-flops. The certainty that is incoming to the D port of a flip-flop will subsequently propagate forward ($\overrightarrow{fwd}$) through the output logic cone of the Q port. Likewise, the incoming certainty received at the Q port will propagate backwards ($\overleftarrow{bkwd}$) through the input logic cone of the D port. Figures 4.8(b)-(c) illustrate this sequence of events. In this figure the green/dashed arrow labeled with **1** is the first propagation, which subsequently causes the second propagation to occur (labeled with **2** and shown in red/solid arrow). Algorithm 2 describes this sequence in pseudo-code format. Note, each request *REQ* contains more information (see Figure 4.7(a)), however we highlight only the fields that are essential for explaining Algorithm 2 (the same applies to Algorithms 3 and 4). If multiple propagation requests are received by a flip-flop, only the one that has the highest *certainty* so far will be taken into account. Note, the incoming D and Q values do not affect each other and are stored separately, so that bit-flip detectability can be correctly computed, as formulated in section 4.3.1.

A traced flip-flop, as the original source of certainty, generates two propagations (on each port), as shown in Figure 4.8(a). The traced flip-flops are *fully* Q-Restorable, since their output is connected to the trace memory. Hence, we can set their Q-certainty to 1. This knowledge must then be propagated both forward and backward. For example, if we trace only the flip-flop `B` in Figure 4.2(a), the next sink element (flip-flop `C`) receives full certainty at its D port and the previous source element (flip-flop `A`) receives full certainty at its Q port. Note, there is no implicitly available

Algorithm 2 Certainty Propagation through a Flip-Flop

```
Data: Certainty already stored in this flip-flop: CV_D, CV_Q
1: procedure RECEIVE-FLIP-FLOP(port, CV_in)
2:     if port = D then                                   ▷ Incoming port is D
3:         if CV_in improves CV_D then
4:             CV_D updates to new CV_in
5:             PPG-ENQUEUE(CV_D, fwd→, fanout nets of Q)
6:     else                                               ▷ Incoming port is Q
7:         if CV_in improves CV_Q then
8:             CV_Q updates to new CV_in
9:             PPG-ENQUEUE(CV_Q, ←bkwd, source net of D)
```

Table 4.1: Backward Propagation from Output to Inputs

Logic Gate	$R1_A$	$R0_A$
$Y = A \cdot B$	$R1_Y$	$R0_Y \cap R1_B$
$Y = A \cdot B \cdot C$	$R1_Y$	$R0_Y \cap R1_B \cap R1_C$
$Y = A + B$	$R1_Y \cap R0_B$	$R0_Y$
$Y = A + B + C$	$R1_Y \cap R0_B \cap R0_C$	$R0_Y$
$Y = \texttt{NOT}\ A$	$R0_Y$	$R1_Y$

certainty for the D port of the traced flip-flops (these values must be computed through certainty propagation from other traced flip-flops).

Rule 2 for Logic Gates

For logic gates, the incoming certainty on the output may only back-propagate to all its inputs. The incoming certainty on any input will forward-propagate to the output port and it will backward-propagate to its other side inputs. Similar to flip-flops all propagations only occur if there is an improvement with respect to the previously received certainty. Unlike flip-flops, logic gates do not propagate an exact copy of the received certainty, but they modify it in accordance with their logical function,

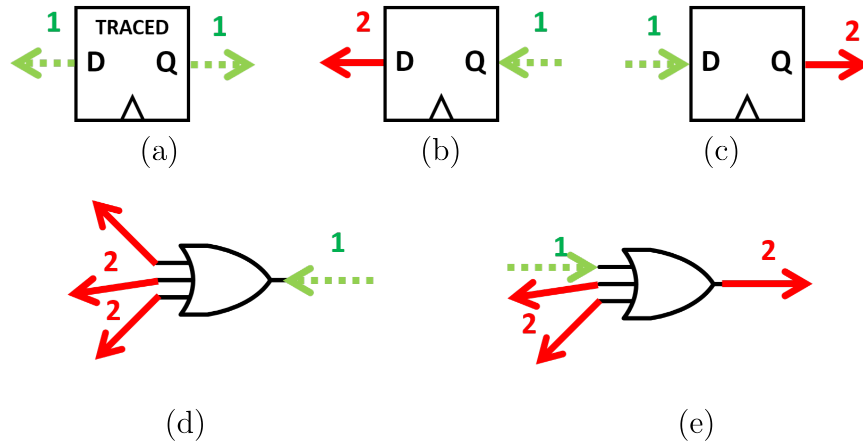


Figure 4.8: Various types of propagation: "**1**" represents the initial request which causes another propagation "**2**". (a) Trace flip-flop initiates forward and backward propagation. (b) Backward to Q-port is echoed back from D. (c) Similarly, forward to D-port is echoed forward through Q. (d,e) Similar to (b) and (c) but for a logic gate.

as shown in Tables 4.1 and 4.2 (using the port labels shown in Figure 4.6(b) for a 2-input AND gate). Note that since backward propagation to input A depends on output Y and the other input B, any received certainty on B (forward from another source) or Y (backward from another sink) triggers a backward propagation to A, as shown in Algorithm 3 and Figure 4.8(d)-(e).

Note, Tables 4.1 and 4.2 describe the propagation rules for logic gates, using the

Table 4.2: Forward Propagation from Inputs to Output

Logic Gate	$R1_Y$	$R0_Y$
$Y = A \cdot B$	$R1_A \cap R1_B$	$R0_A \cup R0_B$
$Y = A \cdot B \cdot C$	$R1_A \cap R1_B \cap R1_C$	$R0_A \cup R0_B \cup R0_C$
$Y = A + B$	$R1_A \cup R1_B$	$R0_A \cap R0_B$
$Y = A + B + C$	$R1_A \cup R1_B \cup R1_C$	$R0_A \cap R0_B \cap R0_C$
$Y = \texttt{NOT}\ A$	$R0_A$	$R1_A$

intersection "∩" and union "∪" notation. Finding the exact values must account for the signal correlations between groups of signals, which may be compute-intensive and hence practically infeasible. In our current implementation we assume that signals are independent and therefore we use approximations, such as $p(X \cap Y) \approx p(X) \times p(Y)$. This results in faster computation at the potential cost of a loss in accuracy.

Algorithm 3 Certainty Propagation through Logic Gates

```
    Data: Certainties already received: CV_Y, CV_A, CV_B,CV_C, ···
    Input: port is one of {Y,A,B,C,...}. CV_in is received certainty tuple.
 1: procedure Receive-Gate(port, CV_in)
 2:     if port = Y then
                                              ▷ Received an update from output
                                              ▷ Must back-propagate to every input
 3:         if CV_in improves CV_Y then
 4:             Update CV_Y to the new CV_in
 5:             for all i ∈ inputs do
 6:                 Calculate CV_out of i using Table 4.1
 7:                 Ppg-Enqueue(CV_out, ←bkwd, i)
 8:     else
                                              ▷ From an input to all others
                                              ▷ Must do both back/fwd propagation
 9:         if CV_in improves CV_port then
10:             Update CV_port to the new CV_in
11:             Calculate CV_out using Table 4.2
12:             Ppg-Enqueue(CV_out, →fwd, Y)
13:             for all i ∈ side-inputs of port do
14:                 Calculate CV_out of i using Table 4.1
15:                 Ppg-Enqueue(CV_out, ←bkwd, i)
```

Rule 3 for Multiple Fan-Out Points

In Figure 4.6(c) K drives more than one other element, namely L and M. These elements can be any gate or flip-flop without affecting the following explanation. Intuitively, forward propagations from K must go to both L and M. Also since any

restoration for L has an effect on M, backward propagations from L to K will also trigger a forward propagation towards M. Similar to the previous section, we assume the signals of a fan-out point are not correlated.

In the example of Figure 4.6(c), the certainty of nets K and L, forward from K and backward from L, must be combined in order to calculate the certainty of M. As shown in Figure 4.6(d), assuming the signals are uncorrelated, one way to compute the combined certainty is $C_M = C_K \cup C_L \approx C_K + C_L - C_K \cdot C_L$. Average value V can also be estimated by the average of all the other V values weighted by their corresponding certainty, $V_M = V_K \times C_K + V_L \times C_L$.

Generally in any split point, containing one stem and n branches, as depicted in Figure 4.6(e), any propagation received on any of the branches (including the stem) will trigger propagation to the other members of the group. The certainty that each individual branch receives, is calculated from all the members (except the receiving branch itself, called *Except* in Algorithm 4). No self-propagation is used (i.e., the contribution of a branch is not propagated back to itself), which explains the name of function COMBINE-ALL-EXCEPT-ONE in Algorithm 4.

Note, we have observed empirically that, when the number of fan-out branches becomes large, the above approximation loses in accuracy. In such cases an alternative approximation can be used, such as the maximum restorability is passed to all the branches, e.g., $R1_M = Max(R1_K, R1_L)$. In Algorithm 4, the COMBINE-ALL-EXCEPT-ONE function uses either the $\cup$ or the Max operator for combining the restorabilities at a fan-out node (this can be either pre-configured by the user or adaptive, i.e., depending on the number of branches). Figure 4.9(a)-(d) shows

examples of a primary certainty propagation that cause secondary and tertiary consequential propagations. The order of these events are shown using the numbers 1-3. In these examples the propagation received by any branch (or stem) generates more propagations.

Algorithm 4 Fan-out Point Integration

Data: Number of branches n, Certainty already received by branches: stem(CV_0), branches($CV_{1..n}$)
Input: b is between 0 and n. CV_{in} is received certainty.

```
1: procedure Receive-Multi-Fan-Out(b, CV_in)
2:     Update CV_b to the new CV_in
3:     if b = 0 then                                  ▷ Received from the stem
4:         for i = 1 : n do
5:             CV_out =Combine-All-Except-One(i)
6:             Ppg-Enqueue(CV_out, →fwd, branch i)
7:     else                                           ▷ From a branches
8:         CV_out =Combine-All-Except-One(0)
9:         Ppg-Enqueue(CV_out, ←bkwd, stem)
10:        for i = 1 : n do
11:            if i ≠ b then
12:                CV_out =Combine-All-Except-One(i)
13:                Ppg-Enqueue(CV_out, →fwd, branch i)

1: function Combine-All-Except-One(Except)
2:     R_1 = R_0 = 0
3:     for i ∈ branches do
4:         if i ≠ Except then
5:             R_1 = Max(R_1, branch[i].R_1)
6:             R_0 = Max(R_0, branch[i].R_0)
7:     C = Min(1, 1 − R_1 − R_0);                     ▷ C Saturates to 1
8:     V = R_1 ÷ (R_1 + R_0)
9:     CV = (C, V)
10:    Return CV
```

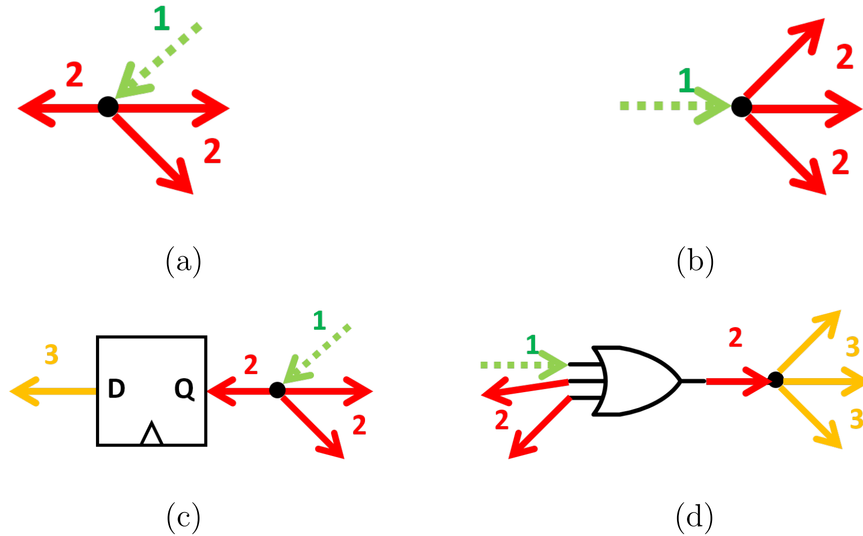


Figure 4.9: Various types of propagation in presence of branches. (a),(b) Illustration of Rule 3: A backward propagation on a branch causes forward propagation to other branches as well as backward to the stem. (c),(d) Examples of combination of Rule 3 with Rules 1 and 2.

The Stopping Criteria

Certainty propagation stops when the propagation queue ($PropQ$) becomes empty, or equivalently when no new requests are generated:

- When the receiving net is a primary input or output, since it is meaningless for the certainty to propagate any further.

- When the receiving net has already received a certainty value that is not greater than the one received previously by former propagations. In this situation the new certainty will **not improve** the restorability and is not helpful. This idea is described in the form of an **IF** statement that compares the received certainty with the one saved in a net (see line 3 of Algorithms 2 and 3)

- When a loop is detected within the circuit. Each propagation request keeps

Algorithm 5 Propagation Enqueue

```
    Data: Certainty Propagation Queue: PropQ
 1: procedure PPG-ENQUEUE(CV_in, dir, dest)
 2:     Build "Request" (REQ) data structure of Figure 4.7(a):
 3:         From = net calling this procedure
 4:         To = dest
 5:         Direction = dir
 6:         (Certainty,Value) = CV_in
 7:         V_p = "Visited" list of parent request
 8:         Visited = V_p ∪ From
 9:     if (dest is a primary input)
            OR (dest is primary output)
            OR (From ∈ V_p) then                    ▷ A loop is detected
10:         Discard this request REQ and return
11:     else
12:         Push REQ into propagation queue PropQ
```

track of the history of all the propagations that led to that particular propagation, using the `visited list` field of the data structure for any request. The list is inherited from one request to all its subsequent ones. When a request is sent from a net, the sender net is marked as a visited net. Later if that particular net receives a request, it is prohibited from propagating it further, using the mechanism described in Algorithm 5. This ensures that propagation traverses the loop for one complete turn, before preventing new requests. This idea is shown in figure 4.10(a) in which propagation (1) starts at flip-flop `A`, (2) goes through the logic gate, reaches flip-flop `B` and (3) propagates forward. In the third step, the loop is detected and the request only updates the D-restorability of flip-flop `A`, but cannot go any further. In Figure 4.10(b) the source of the propagation is from the outside of the loop. The logic gate in the middle of the flip-flops receives a request from one of its inputs and generates two propagation

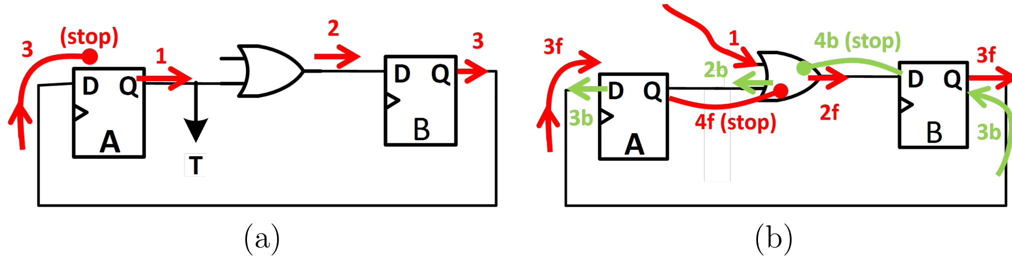


Figure 4.10: Propagation stops when a complete loop has been traversed. (a) In the third step the loop is detected. CV_D is updated and propagation stops. (b) Initial request causes two propagations, backward and forward, 2b and 2f. Each circulates for one loop and stops.

requests in both forward and backward directions (2b and 2f). Further propagations (3b and 3f) circle the loop, finally 4b and 4f close the loop and stop the propagation. Otherwise the logic gate would generate more unnecessary requests.

4.3.3 The Heuristic for Trace Signals Selection

We will now describe the top-level heuristic for trace signals selection, which works based on the previously introduced metrics and certainty propagation rules. In Algorithm 6, T is the number of trace signals to be selected (trace budget) given by user and n is the number of flip-flops in the circuit. As described in the algorithm, the outer loop of the code runs T times and in each iteration it selects one new trace signal. Within the inner loop, we need to decide which of the unselected flip-flops is the best candidate to be added to the list of trace signals ($\mathbb{L}$).

In the inner loop (lines 7:9 of Algorithm 6), we consider one flip-flop at a time and compute its incremental impact on the bit-flip detectability (BFD), which was defined at the end of section 4.3.1, of all the flip-flops in the design. To achieve this,

Algorithm 6 Trace Selection Algorithm

Data: Trace count (T), Flip-flop count (n)
Result: Set of traced flip-flops ($\mathbb{L}$)
1: **function** SELECT(T, n)
2: $\mathbb{L} = \emptyset$ ▷ Empty set
3: **for** $t \leftarrow$ 1:T **do**
4: Define Array Scores[n] = {0,0,...,0}
5: $S1$ = COMPUTEBFD($\mathbb{L}$)
6: **for** $i \leftarrow$ 1 :n **do**
7: **if** $Flop_i \notin \mathbb{L}$ **then**
8: $S2$ = COMPUTEBFD($\mathbb{L} \cup$ $Flop_i$)
9: Scores[i] = $\sum_{k=1}^{k=n}(S2[k] - S1[k])$
10: BestFlop = Flip-flop i that maximizes Scores[i]
11: $\mathbb{L} = \mathbb{L} \cup$ BestFlop
12: **Return** $\mathbb{L}$

each flip-flop that is not selected so far is temporarily counted within the selection set and the bit-flip detectability of all flip-flops are recomputed by Algorithm 7. This algorithm returns a vector of n values, one BFD for each flip-flop. This function sets the Q-certainty of all the traced flip-flops to 1 and assumes an equal chance of observing zero and one for the traced flip-flops, $V = \frac{1}{2}$, unless otherwise provided by the user [1] (e.g. prior knowledge from simulation). It then triggers the propagation from the traced flip-flops, in both directions, in accordance to the rules described in section 4.3.2.

Line 9 of Algorithm 6 computes the improvement in BFD of each flip-flop as the result of including a new flip-flop within the selection of trace signals ($Flop_i$). This difference does not take the baseline of the improvement into account. For example an improvement from 0.3 to 0.4 is seen exactly the same as 0 to 0.1. This lack of distinction may result in high BFD for some flip-flops and BFD starvation for

[1] Experiments with $V \neq \frac{1}{2}$, where V would be extracted form simulation dumps, showed no significant difference in the outcome of the selection algorithm and/or the quality of bit-flip detection.

Algorithm 7 Compute Global BFD of a Selection

Data: A List of Selected Signals ($\mathbb{L}$), Flip-flop count (n)
Result: Bit-Flip Detectability Score S for each element of $\mathbb{L}$

1: **function** COMPUTEBFD($\mathbb{L}$, n)
2: Define S as array of size n
3: $S = \{0, ..., 0\}$ ▷ Initialize Score to Zeros
4: Reset certainties of all nets to Zero
5: **for** $i \in \mathbb{L}$ **do**
6: $CV_Q = (C = 1, V = \frac{1}{2})$
7: Set-Certainty of $Flop_i.Q$ to CV_Q
8: PPG-ENQUEUE(CV_Q, $\overrightarrow{fwd}$, fanout nets of Q)
9: PPG-ENQUEUE(CV_Q, $\overleftarrow{bkwd}$, source net of D)
10: PROPAGATE-ALL
11: **for** $i \leftarrow 1$:n **do**
12: Compute BitFlipDetectability $S[i]$ for $Flop_i$
13: $S[i]$ = QR1×DR0 + QR0×DR1
14: **Return** S

others. By substituting $\sum_{k=1}^{k=n}(S2[k] - S1[k])$ with $\sum_{k=1}^{k=n}(\frac{S2[k]-S1[k]}{1+\alpha S1[k]})$, we will have a non-linear subtraction that can reward improvements of lower BFD flip-flops. For instance with $\alpha = 10\%$ the improvement from 0.3 to 0.4 is scaled down to the same as 0 to $0.077 = 0.1/(1 + 0.3)$. By using this parameter α we can prioritize the improvement for the less detectable flip-flops.

4.4 Results

In this section we present results based on bit-flip injection experiments. We first discuss our experimental setup, which is followed by an evaluation of bit-flip detectability and diagnostic resolution. Finally, we assess the impact of different parameters for the trace signals selection algorithm.

4.4.1 Experimental Setup

In order to evaluate the effect of the selected trace signals on bit-flip detectability, we first needed to design a (virtual) evaluation platform for injecting bit-flips into any of the flip-flops in a test circuit. We parse the input netlist and at the output of every flip-flop add a multiplexer that can output $\overline{Q}$ instead of the Q, thus injecting a bit-flip at any configurable time. For a circuit with n flip-flops we ran n simulations and each time we injected a bit-flip in a distinct flip-flop. We then repeated this experiment with a different trace budget. For example, each data point in figure 4.12 represents one such experiment. After an initial state of all flip-flops reset to zero followed by 10,000 warm-up cycles, traces are collected for a window of 100 cycles, in which random stimuli are applied to the circuit, with the following exceptions. Unlike another recent work in which the last state of circuit is also dumped (possibly through scan chains) [78], here we only rely on the contents of the trace buffer to detect bit-flips.

We applied deterministic patterns to the few synchronous reset and control signals from circuits `s35932` and `s38584` (from the ISCAS89 benchmark set [93]), similar to what has been done in [95]. For `s35932`, signals {TM1, TM0} are driven to {1,0} (a more detailed analysis on `s35932` will be provided in the following section). Bit-flips are injected in the middle of the trace window. The collected traces together with the circuit structure are converted to a Satisfiability instance and are passed to the PicoSAT solver [91]. From the core of UNSAT of every unsatisfiable instance we can extract a list of suspect flip-flops and clock cycles which play a role in the logic inconsistency. The trace signal selection algorithm from section 4.3 is used with the α parameter set to 10%. Also, when combining certainty at multiple-fan-out points,

we used the $\cup$ operator if the output fan-out branches are less than or equal to 3, and the Max operator otherwise. A final point worth mentioning is how we have defined the "improvement of certainty" as used in Algorithms 2, 3 and 4. Ideally certainty of a net **improves** from C_1 to C_2, only if $C_2 > C_1$. However, in order to speedup the computation and avoid queue explosion, this condition was relaxed to $C_2 > (C_1 + \Delta)$ in which Δ is a small positive number configurable by the user (by default $\Delta = 0.001$ unless otherwise stated).

4.4.2 UNSAT Rate: Bit-flip detectability

Figure 4.12 shows the percentage of the UNSAT problems when collecting a varying number of trace signals, for the three largest ISCAS89 benchmark circuits[93], containing approximately 1,500 to 1,700 flip-flops and more than 20,000 nets. The horizontal axis is the percentage of the flip-flops that are traced. We define a flip-flop as *"Covered"* if after multiple bit-flip injections, at least one is detected. The right part of figure 4.12 shows the number of covered flip-flops for 5 injections. We believe this covered flip-flops metric is relevant since in post-silicon validation experiments, which run for extensive durations (hours to days), if at least one bit-flip in the same flip-flop is detected and analyzed it can provide meaningful feedback information to the design/process. The results show the percentage of detected bit-flips are more than double the percentage of the traced signals. The results for our bit-flip detection driven trace signals selection algorithm are consistently better (by approx. 15%) than for randomly selected trace signals (as done in our exploratory work presented in [96]) or restoration-based method explained later. We attribute this improvement primarily to the introduction of the *"bit-flip detectability"* metric that guides the trace

signals selection algorithm. The significance of bit-flip detectability in comparison with a restoration-based method is discussed later.

Without loss of generality, in `s35932` the mode signals {TM1,TM0} are asserted to {10}. This forces the circuit to remain in a single operation mode for the entire experiment. The results for other modes are similar. Noticeably for `s35932` the detection and coverage rate are significantly higher, and our interpretation is that it is due to its numerous shift register structures. Synthesizing this circuit revealed some insight about its internal structure. Forcing the mode to any of the 4 possible values, reduces the circuit to only XOR gates, inverters and flip-flops mostly configured as shift registers, as summarized in Table 4.3. To explain the observed high detection rate, we rely on a circuit that is shown in figure 4.11. This partial circuit is a simplified version of what can be seen in `s35932`'s structure. As it can be seen in this example, tracing one flip-flop from the top shift register can restore many other bits over multiple clock cycles. For example by recording a history of flip-flops A and B we can anticipate exactly what value to observe on C and consequently D. Furthermore, if we trace D in addition to A and B, we can detect a bit-flip on any of the flip-flops on the path from A to D (including D itself). However this detection suffers from a poor diagnostic resolution, since we cannot localize the bit-flip occurrence to any individual flip-flop or clock cycle, as detailed in the following-subsection.

Table 4.3: Different Configurations for `s35932`

Mode	Shift registers	Total bits	Synthesized circuit contains
0	9x32b + 288x4b	1440	1728x FF, 297x NOT, 1179x XOR
1	9x32b + 288x5b	1728	1728x FF, 313x NOT, 443x XOR
2	9x32b + 288x4b	1440	1728x FF, 297x NOT, 1179x XOR
3	9x160b	1440	1728x FF, 329x NOT, 443x XOR

Figure 4.11: A simplified view of a typical circuit block from `s35932`

In summary, for a circuit like `s35932` with shift registers and XORs: (1) majority of the bit-flips are detected with relatively few trace signals, and (2) the suspect list contains many flip-flops, since we cannot know exactly where and when the bit-flip has occurred.

4.4.3 Core of UNSAT: Size and Time Distribution

As seen in figure 4.12, a wider trace leads to a better detection rate, which is an intuitive result. Another benefit of a wider trace is a reduction in the list of suspect flip-flops. Figure 4.13 shows a histogram of the length of this list for `s38417` and `s35932`. The bar histograms (in green) show the distribution for 256 traces and the steps (in blue) belong to 128 traces. For reasons discussed earlier, suspect lists for `s35932` are larger than `s38417` despite the size of these two circuits being comparable. In terms of timing, figure 4.14 shows how far away are the suspect clock cycles from the time of bit-flip injection. `s38417` peaks at the vicinity of the injection time (shown as time 0 in the chart), while `s35932` has a larger share of suspects from **before** the injection time. Again, this is because of the structure of this circuit and the how

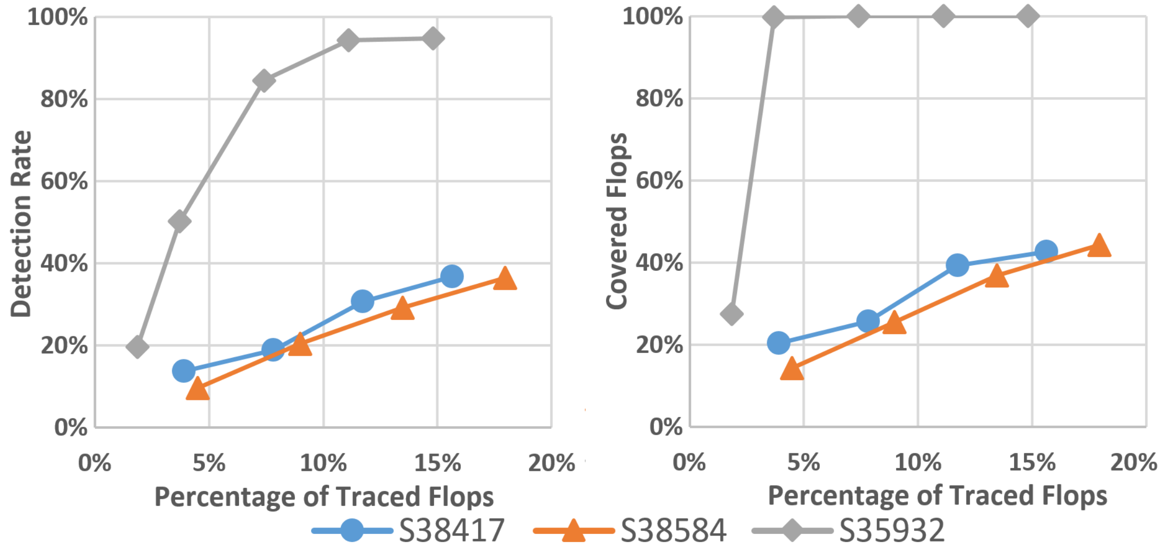


Figure 4.12: Left: Rate of UNSAT problems vs. percentage of flip-flops to trace. Right: Percentage of covered flip-flops after 5 injections.

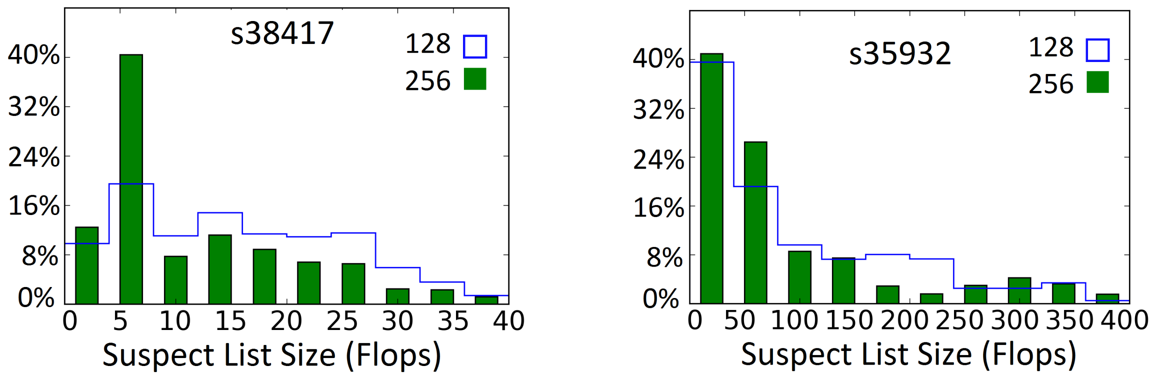


Figure 4.13: Distribution of the size of the suspect flip-flop list in the core of UNSAT.

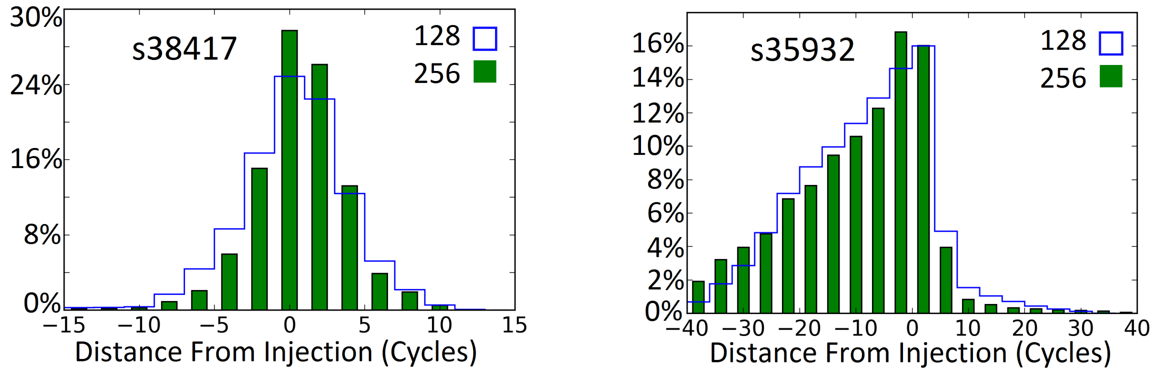


Figure 4.14: The difference in clock cycles between the injection time (t=0) and the time-step of the suspected signals in the core of UNSAT.

many branches converge to form another net through XOR gates. For instance, in order to detect a bit-flip in flip-flop C in figure 4.11 we need to restore it from two sides, i.e. its D-port and Q-port. To restore its D-port, we need knowledge about a number of flip-flops on its transitive input logic cone (similar to A, B and the path from A to C). On the other hand for recovering C's Q-port, we need D and the other input of the XOR (path from A to D). As can be seen, the suspect list of a possible detection on C consists of many flip-flops associated with negative time-steps relative to the time of bit-flip in C (similar to A or B) and fewer flip-flops associated with positive time-steps (similar to D or other nets on C's transitive fanout).

4.4.4 Comparison with Restoration-based Method for Trace Signals Selection

A major motivation of this work is centered around detecting bit-flips by introducing a new metric to drive the selection of trace signals. In order to see the effectiveness of this metric in comparison with the restoration-based methods, we decided to run experiments and compare the two. It is worth noting that there is a fundamental

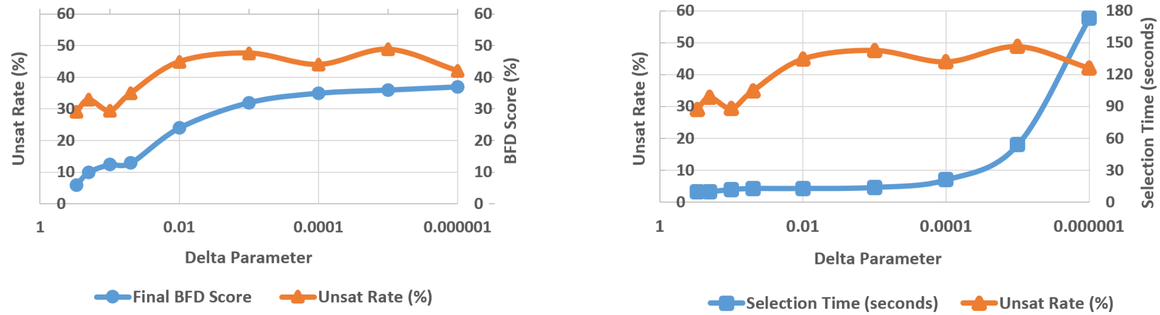


Figure 4.15: The effect of reducing threshold for certainty comparisons for s9234 with $\Delta = \{0.3, 0.2, 0.1 : 10^{-6}\}$.

difference between the objectives of using these two methods, which was illustrated in figure 4.4 from subsection 4.2.1. The restoration-based methods are focused on detecting functional bugs *in the absence of logic inconsistencies*, and hence they aim at restoring the values of unknown bits based on the collected trace. In the presence of electrical bugs, however, a value in a flip-flop gets inverted and therefore the propagation of the traced values might lead to logic inconsistencies. That is why we need to have *multiple sources of information to detect these inconsistencies*, which is totally redundant when selecting the trace signals for restoration purposes only.

In order to perform a comparison, a restoration-based method was implemented in which, similar to our BFD-based selection, certainty of signals propagates throughout the entire circuit with the following differences. Since separate D and Q restorations for flip-flops are removed and replaced with a single certainty value, the BFD formulation (equation 4.1) no longer applies. Hence, each flip-flop is assigned only a single certainty value which is updated when a propagation is received from either of its two ports (D or Q). Note that for the BFD implementation, the certainty received from the D port is stored separately from the one that is received from the Q port.

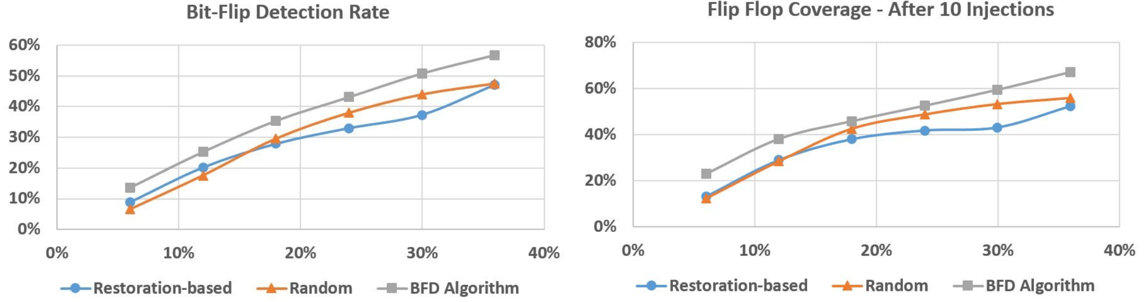


Figure 4.16: Left: Rate of detected bit-flips vs. the percentage of flip-flops to trace. Right: Percentage of covered flip-flops after 10 injections. Circuit under test is `s15850` with 534 flip-flops. A 10-15% improvement is consistently observed for the evaluated circuits.

For selecting the signals (Algorithm 6) instead of optimizing for maximum BFD of all flip-flops (line 13 in Algorithm 7), we aim for maximizing the certainty of all the flip-flops, i.e., the same line simplifies to $S[i]$ = R1 + R0.

Figure 4.16 shows the results for three different selection algorithms based on (1) Restoration, (2) Bit-flip detectability and (3) Random selection for the benchmark circuit `s15850` [93]. As shown in the figure, even random selection of traces can most often outperform the restoration-based selection both in detecting a single bit-flip and in covering at least one-out-of-ten bit-flips. The underlying reason for such a shortcoming is that once a restoration-based method identifies a subset of circuit nodes to be restorable by a group of trace signals, it tries to cover new parts of the circuit using the remaining trace budget. Therefore a restoration-based approach ignores restoring the value in a flip-flop on both of its ports, as needed for the identification of bit flips and which is prioritized by our new bit-flip detectability-driven selection of trace signals.

4.4.5 Assessing Different Algorithm Parameters

There are a number of parameters that can be adjusted for an experiment, such as α (discussed in section 4.3.3) or Δ (section 4.4.1), using the Maximum (Max) or $\cup$ for fan-out point integration or both (Algorithm 4), forcing expected value of traced flip-flops to 50% or measuring them from simulation.

In a set of experiments on `s9234` [93] the Δ parameter has been varied from 0.3 to 10^{-6} while the following were measured as shown in figure 4.15: (1) UNSAT ratio shown in percentage which is the ratio of detected bit-flips (same definition as used in figure 4.12), (2) the "Bit-Flip Detectability" score of a selected trace at the end of the selection process and (3) the runtime of the selection algorithm. As can be seen, a smaller Δ which leads to a more accurate model for distinguishing certainty improvement in a net, eventually results in higher BFD Score. This also confirms the correlation between BFD Score and the quality of the selected trace signals with higher UNSAT ratio for higher BFD score (see left half of figure 4.15). It is worth mentioning that after a certain point ($\Delta \leq 0.001$) the UNSAT ratio is not necessarily improving since it saturates even with higher BFD Score. For $\Delta < 10^{-3}$ the UNSAT ratio fluctuates briefly which is most likely due the random nature of the experiments (i.e. the random stimuli applied to the circuit under test). The right half of figure 4.15 reports the runtime of the algorithm which grows drastically for very small Δ. This is because even a very small (and probably insignificant) improvement in the certainty of a net creates many new propagation requests to other nets which needs additional CPU time for processing, without bringing tangible value to the selection.

Table 4.4 shows 6 different configurations set up for the algorithm. Each configuration leads to a particular UNSAT ratio as shown in the bar graph figure 4.17.

Table 4.4: Different Configurations for the Trace Signals Selection Algorithm

No.	Configuration
1	Random Trace (s38584:256b, s9234:64b)
2	$\cup$ used to merge fanouts, $\alpha = 0.1$
3	Max used for $n \leq 3$, otherwise $\cup$, $\alpha = 0.1$
4	Max used for $n \leq 5$, otherwise $\cup$, $\alpha = 0.1$
5	Max used to merge fanouts, $\alpha = 0.1$
6	Max used to merge fanouts, $\alpha = 0.1$, $V \neq 50\%$

These results belong to two different circuits (s9234 and s38584). As can be seen, different circuits can be sensitive to changes in parameters differently. Note that configuration 1 is the average of 10 random trace selections and is not related to the algorithm. It is merely reported as a point of reference. In C2, the $\cup$ method is used to merge certainty of all branches. In C3, smaller branches (up to 3) are combined with Max method and larger branches use $\cup$. Analyzing the two circuits shows that 75% of branches in s9234 and 63% of them in s38584 have a fanout of 3 or less and are combined with Max method in C3. In C4 more branches are combined with the Max method (roughly 83% for both circuits), and finally in C5 all the branches are combined with Max. The observed trend is that using the $\cup$ together with Max is, as could be predicted intuitively, more effective. In all of the above experiments, when tracing a flip-flop, its "Average Value" or V would be set to 50%, asserting that each signal has an equal chance of being 0 or 1. In contrast for C6, we ran simulation for a million clock cycles and recorded how often each flip-flop has been 0 or 1 and used this information as the starting point for its V value. As can be seen this did not have any visible effect on the UNSAT ratio.

The selection algorithm ran on a Quad-core Intel i7-2600™ CPU in multi-threaded

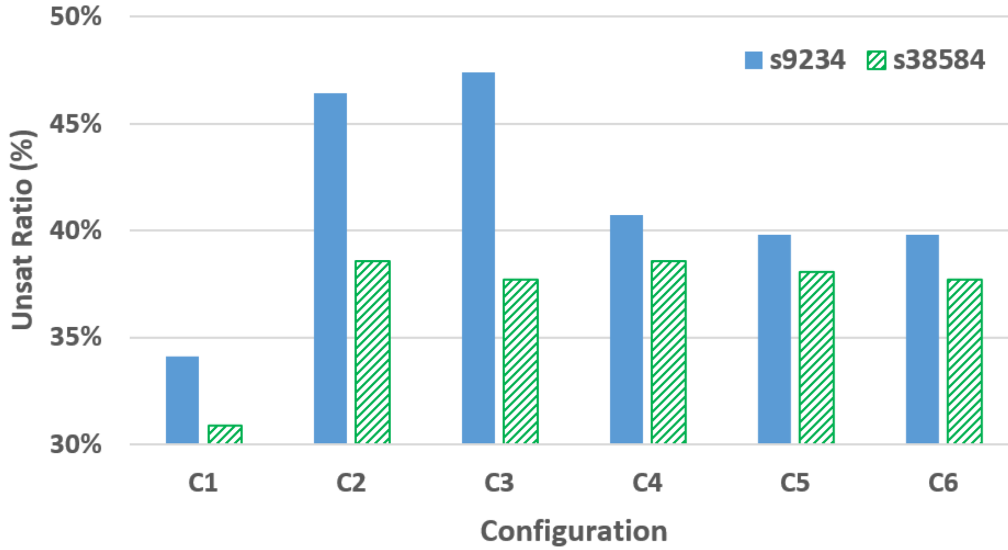


Figure 4.17: Various configurations for selecting signals (see Table 4.4).

Table 4.5: Runtime for the Trace Signals Selection Algorithm (seconds)

Circuit	FF count	Tr 64	Tr 128	Tr 192	Tr 256
s35932	1728	979 s	1465 s	1873 s	2640 s
s38584	1426	798 s	819 s	1181 s	1392 s
s38417	1636	689 s	1165 s	1770 s	2111 s

mode. The trace signal selection algorithm runs in multi-threaded mode (4 threads) with runtimes as reported in Table 4.5 for the three largest ISCAS89 benchmark circuits. While these runtimes can be in range of thousands of seconds, it is worth noting that the trace signals selection algorithm needs to run once at design time.

4.5 Summary

In this chapter the following topics were discussed :

- Restoration-based methods for choosing a set of traced signals do not capture the information required for finding a bit-flip. Those methods can be helpful when dealing with functional bugs when signal consistency is preserved. Electrical bugs, however, may lead to corruption of the Boolean state of signals (bit-flips) and lead to inconsistent logic. More information is needed to detect these inconsistencies.

- A new metric called "bit-flip detectability" (BFD) is introduced which focuses on the probability of detecting bit-flips on flip-flops in a digital circuit.

- A BFD-based algorithm was proposed which tries to maximize the chances of restoring flip-flops to two different values from two different paths, hence detecting a flipped value.

- Details were provided on how such a probability metric is supposed to propagate through the circuit. The algorithm relies on propagation of the probability within a netlist for making its decisions.

- Experiments show that the selected set of traces produced by this algorithm are effective. Random set of traces were used as a base line for comparison. We concluded that Restoration-based method can be less effective than a random decision while the proposed algorithm could consistently outperform both random-decisions and restoration-based methods.

Chapter 5

Joint Selection of Trace Signals and Assertion Checkers

5.1 Background

In the previous chapters, the use of Boolean SAT solvers to detect bit-flips was discussed as well as choosing which signals to trace for improved bit-flip detectability. As shown in Figure 5.1 we assume that an on-chip memory stores an ongoing history of several signals within the system while it is operating. Later the content of that memory is offloaded to be analyzed by an automatic software to firstly detect an issue (in terms of an inconsistent state or Boolean contradiction) and secondly find the root cause of such inconsistency (in terms of a list of suspects that need to be further investigated by the designer). The offload process needs to be initiated when a failure has occurred or an event of interest has happened. As a result an important part of the design-for-validation infrastructure that enables an effective post-silicon validation process, is the event detection mechanism.

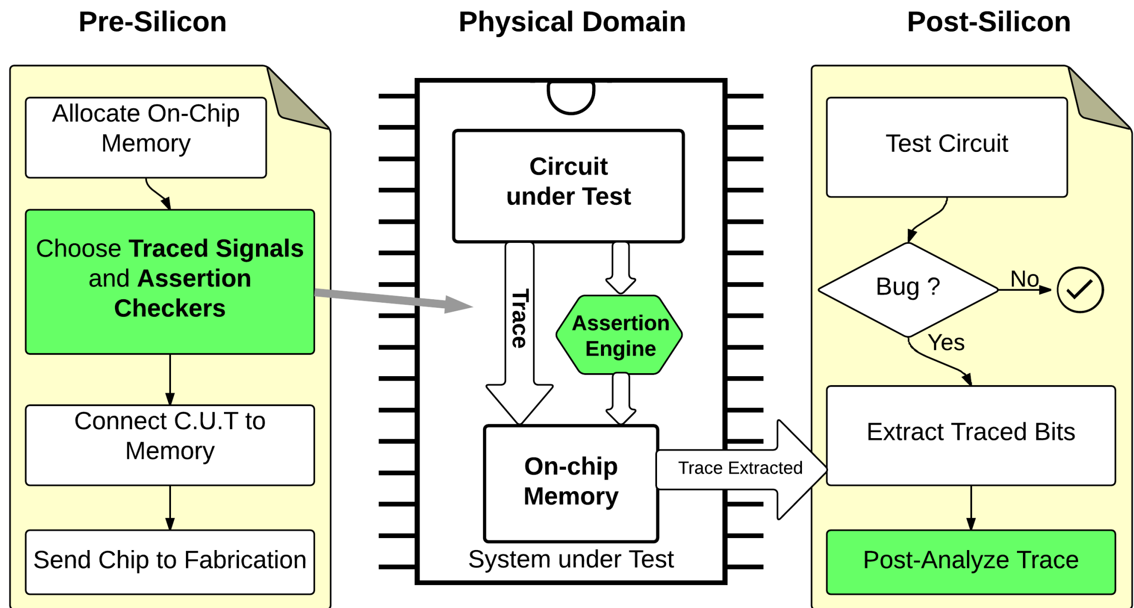


Figure 5.1: Big picture: Pre-silicon and post-silicon tasks for collaborative trace-and-assertion-based debugging. In pre-silicon the desired signals to be traced and the assertion checkers to be synthesized must be selected. During post-silicon validation, the collected trace is extracted as well as the violations caught by the assertion engine to indicate any of the assertions that fired during the debug session. All of this debug information is then post analyzed to generate a list of suspects, useful for root-cause analysis.

Assertion-checkers, which are historically inherited from software development, have been used extensively in pre-silicon verification for over a decade [97]. While on-chip assertions checkers can potentially detect both electrical and functional bugs by monitoring properties in real-time, there are a few challenges to integrate assertion checkers in post-silicon validation. Translating assertions to hardware circuits (assertion synthesis), with applications to online monitoring and in-field diagnosis, has been investigated over the previous decade [98]. Also, time-multiplexing assertions for silicon debugging has been researched in [99]. Nevertheless, considering the large pool of assertions that can be generated automatically, e.g., [100], raises questions concerning how many assertions to use for post-silicon validation and what is their relevance for improving the detection of bit-flips. This problem has been studied recently in [101], however there has been no consideration on how on-chip assertion checkers can complement the validation data collected in trace buffers.

In order to benefit from both hardware assertion checkers and validation data that can be acquired in trace buffers, one can decide at design-time:

- which signals to trace in order to aid the root-causing of bit-flips caused by electrical bugs;
- which assertion properties should be synthesized and integrated on-chip for bit-flip detection.

Both of the above problems have been studied independently from each other. However, the two challenges are related in the sense that both types of methods use hardware resources, such as on-chip area and wires, to find the root cause of unexpected events during post-silicon validation. Since there appears to be no major

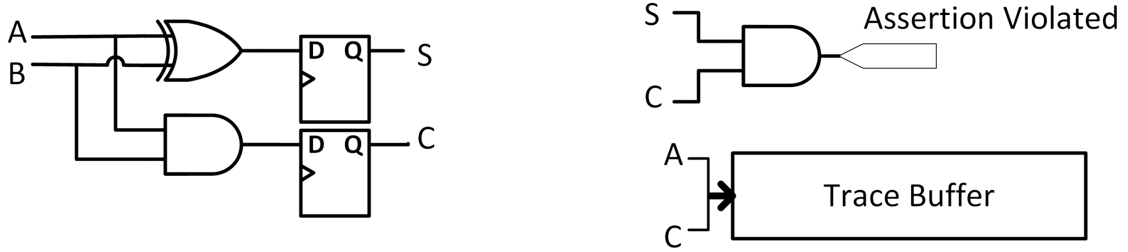


Figure 5.2: A half-adder cell with trace buffer and a single assertion property checker. Assertion checkers get violated when both C and S are logic 1, which is an illegal state for this circuit. Trace buffer is recording a history of only two nets, A and C.

studies that tackles the two types of *selections* at once, we decided to investigate the two problems concurrently during design-time. As demonstrated by our results, using such a *co-selection*-based approach will enable a more efficient usage of hardware resources allocated to trace buffers and assertion checkers in order to **maximize the detection of bit-flips while saving the hardware cost**.

Note that in this work we do not focus on how to generate hardware assertions. We rely on any third party assertion generator tools to provide a set of assertions. All the results in this chapter rely on assertions generated in [102].

5.2 Motivation and Definitions

A simple example is provided in this section to clarify the type of resource sharing between assertion checkers and trace buffers. Definition of the technical terms used in the example are provided at the first usage of every term.

5.2.1 Example

Figure 5.2 shows a half adder cell which consist of two logic gates and four flip-flops. A simple assertion property checker which is `AND` of `sum` and `carry` is also shown which gets violated when both signals are high at the same time. The property checked by this assertion can be described as `‘‘if (C == 1), assert (S==0)’’`

or equivalently:

```
assert (C == 1) => (S == 0)
```

A **bit-flip** (an *undesirable change of Boolean value* due to an electrical bug) on flip-flop `C` from value `0` to `1` is shown as `C` ($\uparrow_0^1$). This assertion checker can detect two bit-flips:

- `C` ($\uparrow_0^1$) when `S` is 1
- `S` ($\uparrow_0^1$) when `C` is 1

If nets `A` and `C` are also traced (i.e. their value is recorded inside the trace buffer), the following two bit-flips may also be detected:

- `C` ($\uparrow_0^1$) when `A` is 0
- `A` ($\downarrow_0^1$) when `C` is 1

By tracing wires `A` and `C` as well as integrating the above assertion checker, we can potentially cover 3 bit-flips while **sharing wire `C` between hardware assertion engine and trace buffer**. Detection through different sources is illustrated in Fig. 5.4.

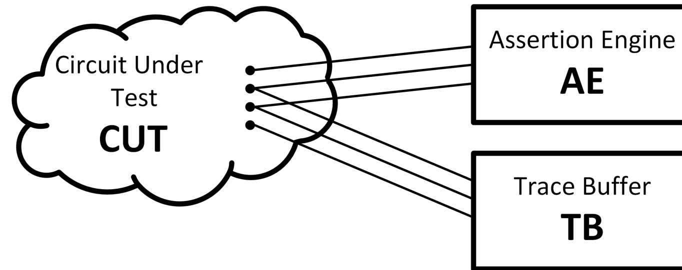


Figure 5.3: Trace buffer (TB) and the assertion engine (AE) collect real-time data from circuit under test (CUT), sharing wires to minimize routing costs. TB is assumed to trace only flip-flops from CUT and wires fed to AE.

5.2.2 General Objective

In a more generic example, as depicted in Figure 5.3, there are (1) a circuit under test (**CUT**), (2) an assertion engine (**AE**) fed with several wires and (3) a traced buffer (**TB**) engine fed with several flip-flops or wires. Wires can be shared between the two engines to reduce routing requirements. The cost of the two engines are explained as:

- **Assertion Engine**: consists of the assertion logic (gates and flip-flops) synthesized to area A units.
- **Trace Buffer**: a memory of depth d and width w to record w signals for d clock cycles. Here we assume this will be synthesized as an on-chip SRAM module.

The assertion unit is accompanied with an *Assertion Data Set* as shown in Table 5.1. This table summarizes the list of all assertions and their cost and benefits, defined as:

- **Cost**: An assertion uses a number of wires listed in the column 2 of Table 5.1. Area cost to implement is in column 3. This cost depends on how many

Table 5.1: Assertion Data Set

Assertion No.	List of Wires	Area Cost	Benefit 1 Covered FF. Bit-Flip Type Detection Rate	Benefit 2 Covered FF. Bit-Flip Type Detection Rate	...
1	$\{W_1, W_2, W_3\}$	100	$(A, \downarrow_0^1, 50\%)$	$(B, \uparrow_0^1, 10\%)$	
2	$\{W_3, W_4\}$	200	$(C, \downarrow_0^1, 20\%)$	...	
...	...				

logic gates and flip-flops are required to capture the event of interest for that particular assertion.

- **Benefit**: Each assertion covers one or more different bit-flips. The covered bit-flips are listed in the rest of the columns. A "benefit" consists three pieces of information: Covered flip-flop F_x, type of bit-flip ($\downarrow_0^1$or $\uparrow_0^1$) and **detection rate**. In order to find this *rate* one needs to run a series of experiments and inject a total of n bit-flips into each of the flip-flops in a CUD. If an assertion is violated for m times during such experiment the detection rate is calculated as m/n.

5.2.3 Key Insights

Figure 5.4 shows a Venn diagram of all possible bit-flips within a circuit, which includes a flip from zero to one or from one to zero for each individual flip-flop. In this diagram a bit-flip is assumed to fall into one of the 4 possible outcomes below:

1. A bit-flip causes a violation in an assertion and is therefore detected;

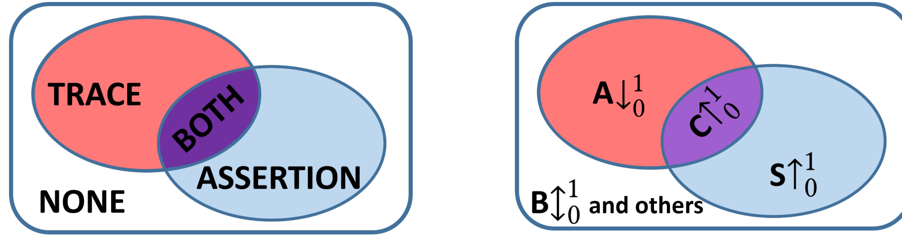


Figure 5.4: Left: Figurative diagram of detecting bit-flips. All the possible bit-flips (every flip-flop, both $\downarrow_0^1$and $\uparrow_0^1$) might be detected by either violating an assertion checker, or with the help of several traced signals, or both, or in the worst case scenario stay undetected. Key insight in this work is to reduce the overlap and spend the budget to cover more bit-flips. Right: Same idea for circuit in Fig. 5.2

2. A bit-flip leaves a detectable footprint in the trace buffer and is therefore detected;

3. A bit-flip causes both (1) and (2), therefore detected;

4. A bit-flip does not cause any footprint or assertion violation and stays undetected.

This work has the following two targets:

- **Discourage detection overlap:** If a bit-flip is detected through one source of information there is little gain in spending budget on the second source,

- **Encourage shared wires:** It is beneficial for assertions to share wires, to minimize cost of routing and not to interfere with the circuit's timing analysis.

The cost function in the following section is designed in a way that gears towards both objectives.

Algorithm 8 Trace and Assertion Co-Selection Algorithm

```
    Data: Trace count (T), Assertion wire budget (W),
          Assertion Data Set (table 5.1)
    Result: Set of traced flip-flops and wires (L_T),
          Set of selected assertions (L_A)
 1: function Select(T, W)
 2:     L_T = ∅                                         ▷ Empty set of traced signals
 3:     L_A = ∅                                         ▷ Empty set of selected assertions
 4:     rT = T                                          ▷ Remaining trace budget
 5:     rW = W                                          ▷ Remaining wire budget
 6:     while (rT > 0) OR (rW > 0) do
 7:         S1 = ComputeBFD(L_T, L_A )                  ▷ current BFD
 8:         C_F = GetFlopCandidates(L_T, rT)
 9:         C_W = GetWireCandidates(L_A, L_T, rT)
10:         C_A = GetAssertionCandidates (L_A, rW)
11:         C_all = C_F ∪ C_W ∪ C_A
12:         Define associative array "Scores"[∀c ∈ C_all] = {0,0,...,0}
                                                        ▷ Zero initial score of all candidates
13:         for c ∈ C_A do                              ▷ Candidate Assertions
14:             S2 = ComputeBFD(L_T, L_A ∪ c)
15:             CF = CostFunction(c, L_A)
16:             ΔBFD = Σ_{k=1}^{k=n} (S2[k] − S1[k])
17:             Scores[c] = ΔBFD / CF
18:         for c ∈ (C_F ∪ C_W) do                      ▷ Candidate nets
19:             S2 = ComputeBFD(L_T ∪ c, L_A)
20:             ΔBFD = Σ_{k=1}^{k=n} (S2[k] − S1[k])
21:             Scores[c] = ΔBFD
22:         Best = Find candidate i whose Scores[i] is maximum
23:         if Best is an Assertion then
24:             rW = rW− NewWires(Best, L_A)
25:             L_A = L_A ∪ Best
26:         else                                        ▷ B is a single net
27:             rT = rT − 1
28:             L_T = L_T ∪ Best
    Return L_A, L_T
```

5.3 Co-Selection Algorithm

5.3.1 Main Algorithm and Cost Function

Algorithm 8 is the main search function. It relies on several other functions which are explained through either words or pseudo-codes. It processes the netlist of the digital circuit and selects a number of traced signals and assertion checkers. Input to the function is the number of traced signals, T, to be chosen as well as the number of wires, W, used for all the selected assertions. Another given information is the data set on all assertions, similar to what is shown in table 5.1. The output of the algorithm are two sets denoted as $\mathbb{L}_A$ for list of assertions and $\mathbb{L}_T$ for list of traced signals. Both of these sets start from an empty state. The remaining budget for both traced signals (rT) and wires for assertion checkers (rW) are checked throughout the execution of the algorithm.

At the core of the algorithm we need to quantify the effectiveness of a certain selection of traced signals and assertions. This is done through the ComputeBFD function which receives the two sets $\mathbb{L}_A$ and $\mathbb{L}_T$ and returns a $1 \times n$ vector, where n is the number of flip-flops. Each element of this vector is a probability of detecting the bit-flip for the respective flip-flop. While any appropriate probabilistic method can be plugged into this function, here we have relied on the probability methods discussed in the previous chapter (i.e. Certainty and Bit-flip detection). Line 7 in algorithm 8 stores the current detection probability of all the flip-flops to be used as a baseline to measure improvements later in lines 16 and 20.

Next, various candidates are listed by three functions, before a decision can be reached:

- $\textsc{GetFlopCandidates}(\mathbb{L}_T, rT)$ provides the list of the flip-flops which are not yet selected. Returns empty if the remaining trace count rT is depleted. Assuming the $\mathbb{F}$ is the set of all flip-flops in a circuit:

$$\textsc{GetFlopCandidates}(\mathbb{L}_T, rT) = \begin{cases} \mathbb{F} - \mathbb{L}_T, & \text{if } rT \geq 1 \\ \emptyset, & \text{if } rT = 0 \end{cases}$$

- $\textsc{GetWireCandidates}(\mathbb{L}_A, \mathbb{L}_T, rT)$ provides the list of wires used in all the assertions listed in $\mathbb{L}_A$. This should exclude all the trace signal already selected and listed as $\mathbb{L}_T$. Also returns empty if rT is already depleted.

$$\textsc{GetWireCandidates}(\mathbb{L}_A, \mathbb{L}_T, rT) = \begin{cases} \text{wires}(\mathbb{L}_A) - \mathbb{L}_T, & \text{if } rT \geq 1 \\ \emptyset, & \text{if } rT = 0 \end{cases}$$

- $\textsc{GetAssertionCandidates}$ $(\mathbb{L}_A, rW)$ lists all the assertion that have not been selected so far. It excludes assertions which require a number of new wires greater than rW. Returns empty when rW reaches zero. Assuming $\mathbb{A}$ is the set of all available assertions:

$$\textsc{GetAssertionCandidates}(\mathbb{L}_A, rW) =$$
$$\begin{cases} \{a \in (\mathbb{A} - \mathbb{L}_A); |\text{NewWires}(a)| \leq rW\}, & \text{if } rW \geq 1 \\ \emptyset, & \text{if } rW = 0 \end{cases}$$

in which "NewWires" is a function that counts how many of the wires used in a particular assertion, a, will be new, considering that some wires may have already

been selected or used by the previously selected assertions.

After listing the candidate signals and assertions, they are evaluated one-by-one and their respective improvement in the bit-flip detection probability is computed in lines 16 and 20. Then the best candidate (either an assertion or a traced signal) is selected such that it leads to better improvement in detection probability. Finally the respective counter (rT or rW) is updated to keep track of the *remaining* budget for each resource.

Comparison Between Algorithms 6 and 8

- Both algorithms are the main search routines of their our task, either choosing trace signals only, or co-selection of trace and assertion. Algorithm 8 is an expansion of 6.

- Algorithm 6 iterates for a predefined number of times through a for loop, while algorithm 8 uses a *while* loop since the number of iterations is not known in advance. Each assertion may use several number of wires and consume the wire budget at a variable rate.

- In algorithm 6 the candidates are simply all the flops, excluding those that are already selected ($Flop_i \notin \mathbb{L}$), while the candidates in algorithm 8 have to be listed in a relatively more complex manner.

- Both algorithms rely on BFD, either from traced signals or from assertion checkers to choose the locally best option.

Cost Function

The cost of a new trace signal is a single wire being added to the selection set. The cost of a new assertion on the other hand depends on how many new wires it requires (which also depends on the previously selected assertions) and the circuit area of that assertion. These parameters are computed by CostFunction as shown in Algorithm 9. Here, the function NewWires lists all the wires required to synthesis all the assertions that are selected so far (W_{all}) and then finds that a specific assertion (A_n) requires only W_{new} new wires. In other words, W_{new}, which is subset of the wires for A_n, provides only the new wires to be added if A_n is eventually selected.

In the CostFunction, two different costs are evaluated and then a single normalized cost is generated. In the experiments in this chapter we assume that the function $x = \text{NORMALIZE}(C_x)$ is a linear function such that:

$$\text{NORMALIZE}(min(C_x)) = 1$$

$$\text{NORMALIZE}(< Cx >) = 2$$

where $< Cx >$ is the average of all costs for the entire assertion database.

5.3.2 Bit-Flip Detection Probability

In order to quantitatively measure the probability of detecting a bit-flip, we have relied on the definitions of "certainty" and "bit-flip detectability" from the previous chapter. The application of bit-flip detectability has been expanded to integrate the detection information through assertion checkers into the model.

The following is a brief summary of the way in which Certainty and BFD measure

Algorithm 9 Cost Function

Data: The new assertion A_n,
list of already-selected assertions $\mathbb{L}_A$

1: **function** CostFunction($A_n, \mathbb{L}_A$)
2: C_A = look up area cost of A_n from data set (table 5.1)
3: $C_W = |\text{NewWires}(A_n, \mathbb{L}_A)|$
4: C = normalize(C_A) + normalize (C_W)
Return C

1: **function** NewWires($A_n, \mathbb{L}_A$)
2: W_{all} = set of all the wired used by all assertions in $\mathbb{L}_A$
3: W_{A_n} = set of all the wires used by A_n
4: $W_{new} = \{c \in W_{A_n} | c \notin W_{all}\}$
Return W_{new}

the probability of detecting bit-flips. The interested reader may refer to the previous chapter for more details.

- The probability of restoring any net in a digital circuit is defined as Certainty, a number between 0 and 1.
- The probability of detecting a bit-flip on a particular flip-flop is defined as the product of certainty the D and Q port of that flip-flop for opposite values. This is shown in Algorithm 10 line 18.
- The certianty values defined for each net propagate throughout the circuit based on the netlist. Traced signals are assumed to have 100% certainty (restoration probability). All other nets start with 0%. The only way to gain non-zero certainty is through propagation from a traced signal to other nets.

Algorithm 10 receives as input two sets of selected assertions and signals and estimates the probability of bit-flip detection on all the flip-flops for that particular

Algorithm 10 Compute Global BFD of a Selection

Data: Set of traced flip-flops or wires ($\mathbb{L}_T$)
Set of selected assertions ($\mathbb{L}_A$), number of flip-flops (n)
Result: Bit-Flip Detectability Score S for each flip-flop

1: **function** ComputeBFD($\mathbb{L}_T, \mathbb{L}_A$)
2: Define S as array of size n
3: $S = \{0, ..., 0\}$ ▷ Initialize Score to Zeros
4: Reset certainties of all nets to Zero

5: **for** $A \in \mathbb{L}_A$ **do** ▷ Apply knowledge from Assertion
6: **for** $B \in$ (benefits of A) **do** ▷ As in Table 5.1
7: (TargetFF, Bit-flip type, Detection rate) ← B
8: SetBFD (TargetFF, Bit-flip type, Detection%)

9: **for** Net $n \in \mathbb{L}_T$ **do** ▷ Knowledge from Trace
10: $CV_Q = (C = 1, V = \frac{1}{2})$
11: Set certainty of Net n to CV
12: Ppg-Enqueue(CV_Q, $\overrightarrow{fwd}$, fanout nets of n)
13: Ppg-Enqueue(CV_Q, $\overleftarrow{bkwd}$, source net of n)

14: Propagate-All
15: **for** $i = 1$: n **do**
16: Compute QRx and DRx for $Flop_i$ ▷ 0 and 1-restorability of Q and D.
17: Compute BitFlipDetectability $S[i]$ for $Flop_i$ =
18: $S[i]$ = QR1×DR0 + QR0×DR1
19: **Return** S

Algorithm 11 Setting BFD of a flip-flop

Data: Restoration probability of this flip-flop: CV_D, CV_Q
External zero and one detectability: $E0D, E1D$

1: **procedure** SetBFD(TargetFF, BitFlipType, Rate)
2: **if** BitFlipType is 0 **then**
3: $E0D = E0D+$ Rate
4: **else** ▷ bit-flip type is 1
5: $E1D = E1D+$ Rate
6: Ensure $ExD \leq 1$; Saturate at 1.0 if necessary
7: PotentialBFD = $E0D + E1D$
8: CurrentBFD = QR1×DR0 + QR0×DR1
9: **if** PotentialBFD ¿ CurrentBFD **then**
10: $QR0 = DR1 = \sqrt{E0D}$
11: $QR1 = DR0 = \sqrt{E1D}$
12: $CV_Q = min(1.0, QR0 + QR1)$
13: $CV_D = min(1.0, DR0 + DR1)$
14: Propagate CV forward to fanout nets of TargetFF
15: Propagate CV backward to source net of TargetFF

selection. It assigns 100% probability to traced signals and lets the restoration probability propagate through the circuit. For assertions, as discussed in the next section, the detection rate of each assertion from *assertion data set* (table 5.1) is taken into account. Eventually the certainties assigned to each flip-flop are used in line 18 of Algorithm 10 to calculate bit-flip detectability of each flip-flop.

5.3.3 Integration of Assertion Benefits

Each assertion may detect one or more bit-flips and each flip-flop may be covered by one or more assertions provided to the algorithm. Thus, we need to keep track of how well each flip-flop is covered at any point during the selection procedure. Parameters $E0D$ and $E1D$ are, respectively, *external zero and one detection probability* of each flip-flop and are used for this objective. Initially ExD of all flip-flops are set to zero.

As more assertions are selected, their bit-flip detection capability (see benefits in Table 5.1) is integrated into the ExD parameters of the respective flip-flop. Introduction of two different parameters has been necessary since we distinguish between the two types of bit-flips: $\uparrow_0^1$and $\downarrow_0^1$.

Algorithm 11 integrates the detection rate of an assertion into the ExD parameter of a flip-flop based on the polarity of the bit-flip. It also makes sure that none of the probability metrics exceed 1 at any point. If the external detection probability leads to a higher detection probability than that generated by the traced signals (through QRx and DRx, as defined in Alg. 10), the QRx and DRx parameters are overwritten by the higher rate from ExD. Lines 10 and 11 of Algorithm 11 use square root of ExD parameters to stay compatible with the definition of "Bit-flip Detectability" as in line 18 of Alg. 10. For example if a flip-flop's restoration parameters were

$$QR0 = 0.2 \quad QR1 = 0.3 \quad DR0 = 0.4 \quad DR1 = 0.0$$

the bit-flip detectability would be $(0.3 \times 0.4) + (0.2 \times 0.0) = 0.12$.
If then we integrate an assertion with benefit of (F, $\downarrow_0^1$, 50%):

$$E0D = 0.5; E1D = 0 \Rightarrow \text{PotentialBFD} = 0.5 > 0.12$$

Consequently the parameters will be updated to reflect the greater detectability:

$$QR0 = 0.707 \quad QR1 = 0.0 \quad DR0 = 0.0 \quad DR1 = 0.707$$

5.4 Results

5.4.1 Experimental Setup

Multiple experiments were conducted to measure the quality of the selections computed by the *co-selection* algorithm in detecting bit-flips. Apart from the netlist of a digital circuit, an essential part of the experiments is the library of the assertions which can be provided from any source such as the designer of a system, pre-silicon assertions or generated by various automatic methods such as [100]. The library of assertions are first evaluated using a digital simulator in which a predefined number of bit-flips are injected to each flip-flop. Then, the number of times each assertion has fired for any bit-flip is recorded to form the assertion data set (table 5.1). Assertions that never fired can be dismissed at this stage to simplify the analysis. Then the algorithm runs and co-selects several traced signals and assertions based on the wire budget provided to it. Another set of simulations are then performed using only the selected subset of wires/assertions to confirm the quality of the selection. The recorded trace signals as well as the violated assertions are then passed to a Boolean SAT solver [91] to detect the bit-flip based on the same concepts as in the previous two chapters. For violated assertions another middle step is required before the SAT solver can be used. For example if the following assertion is violated at clock cycle 100, we know that not only B has been 0 in clock cycle 100, but also A has been 1 on clock cycle 98. Therefore one can conclude that $A[98] = 1; B[100] = 0$

$$if(A = 1) \text{ assert } (B = 1) \text{ after 2 clock cycles}$$

Once the Boolean SAT solver runs, the injected bit-flip is either detected or stays

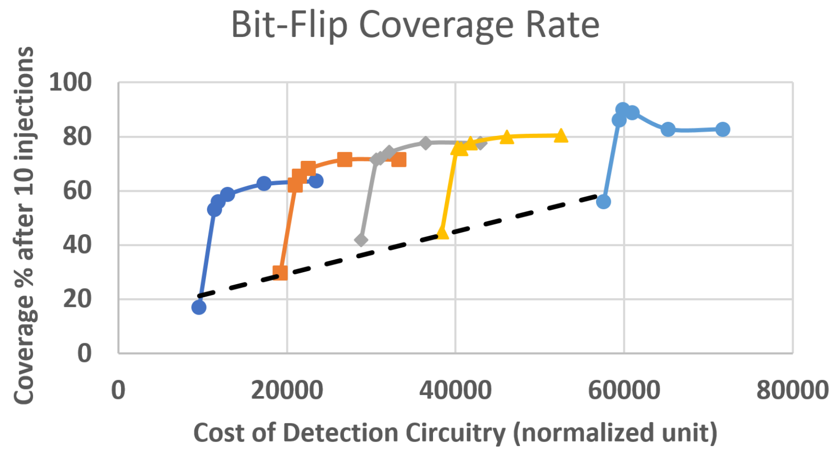
undetected:

- If **not detected**, it shows that the combination of trace signals and assertions could not provide enough information to cause any inconsistency in the Boolean representation of the circuit. SAT solver has found a satisfiable and logical solution for that particular instance of trace and assertion set.
- If **Detected**, we can further analyze the collected trace and find a list of suspect flip-flops by looking at the "Core of Unsatisfiability" generated by the solver. This list of suspects contains the "faulty flip-flop" together with several other nets.

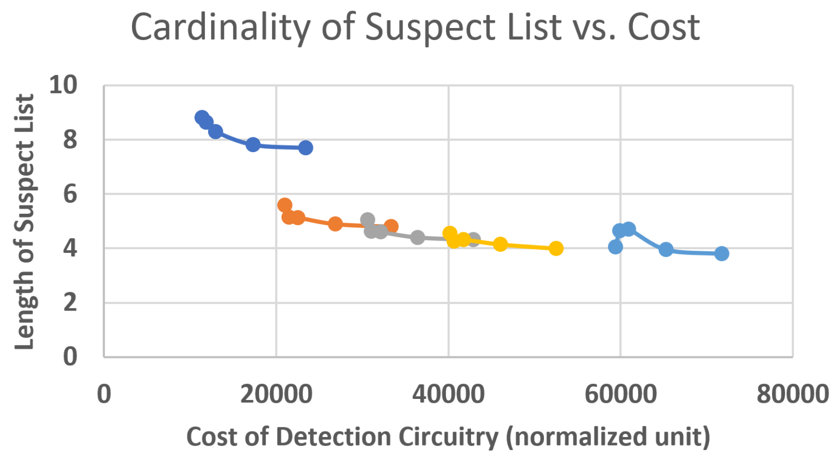
5.4.2 Bit-flip Detection Rate versus Hardware Cost

In this set of experiments we used `s5378` from ISCAS 89. A library of 33,000 assertions were given (automatically generated) for this circuit by [102]. After injecting 10 bit-flips into each of the 179 flip-flops, it was found that almost 10,000 of these assertions never fired. Circuit `s38417` with 598 assertions and 1636 flip-flops is another circuit in this set if experiments.

Results from experiments strongly support the idea that the co-selection method can lead to significantly better detection rate at relatively low extra cost. Figure 5.5.A shows the bit-flip coverage versus the estimated hardware cost for `s5378`. A bit-flip is considered covered if it is detected at least once after 10 injections. Hardware cost is measured as area of the selected assertions when synthesized together with area required for the trace buffer. For assertion synthesis, each flip-flop is assumed to cost 20 units, each 2-input logic gate 8 units and each bit of an SRAM cell 6 units. Trace depth is assumed to be 100 cycles. For example, the lowest data point in Fig.

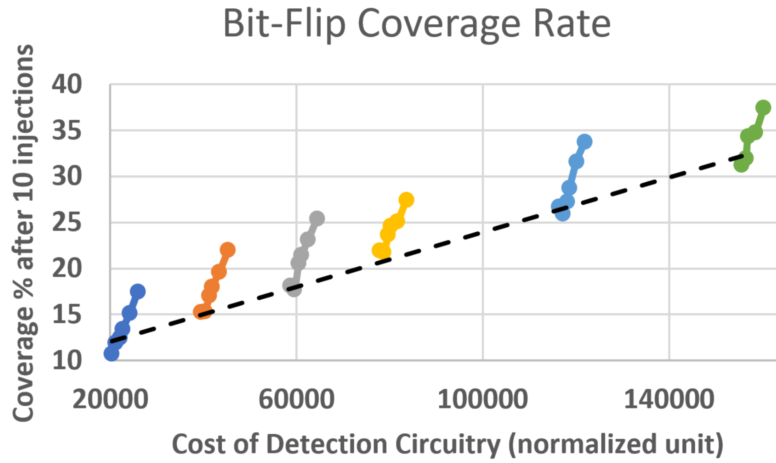


A

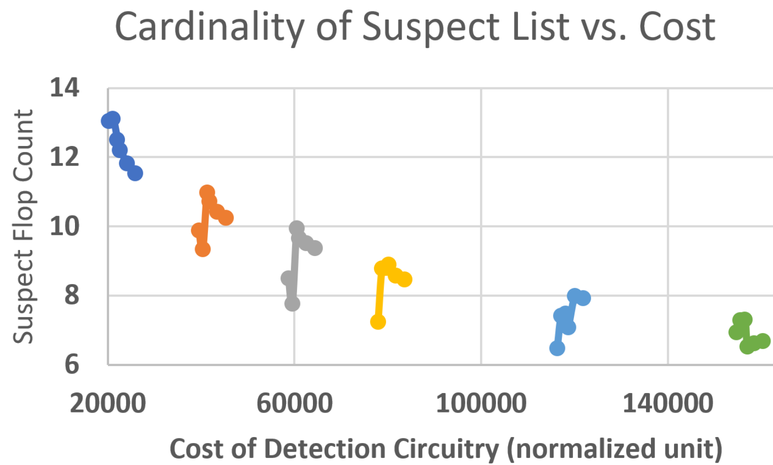


B

Figure 5.5: (A) Bit-flip coverage after 10 injections for `s5378` versus estimated hardware cost. Lower data points connected with a dashed line have no assertion checkers to boost the detection. Each data point is a 10-bit-flip injection experiment. (B) Average count of suspect flip-flops for experiment in A.



A



B

Figure 5.6: Similar experiments as in Figure 5.5 for a different circuit: `s38417`. Bit-flip coverage after 10 injections and average number of suspect flip-flops versus the estimated hardware cost.

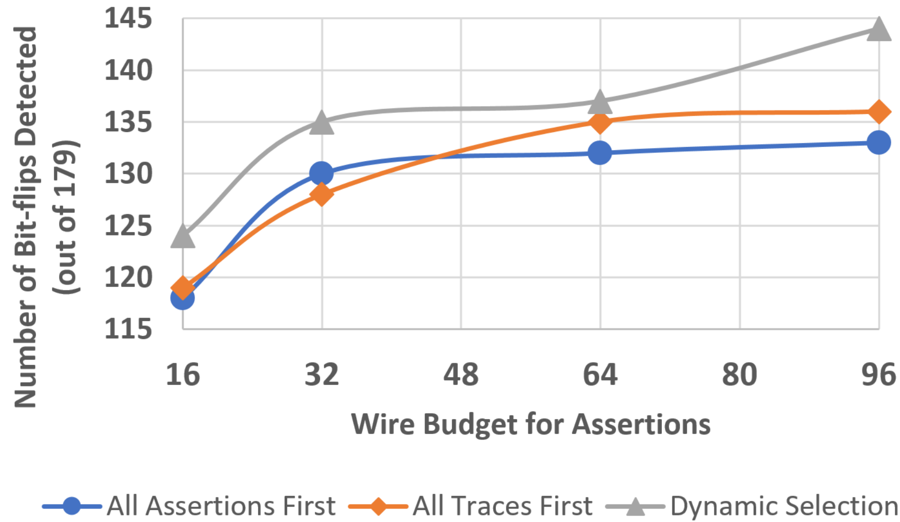


Figure 5.7: Bit-flip detection versus wire-count of the selected set of assertions, for various selection strategies; such as selecting all assertions first and then all the traces. Dynamic selection (co-selection algorithm) performs better.

5.5.A has a cost of 9600 units which originates from only a 16 bit trace buffer, and no assertions ($9600 = 16 \times 100 \times 6$). The sharp increase in bit-flip coverage comes from inclusion of the assertions engine with more wires for each higher data point. The dashed line represents the slope of increase in detection rate, if only trace-buffers were employed. Figure 5.5.B shows the number of suspect flip-flops for the very same experiment. It is shown that list of suspects generally shrinks, with minor exceptions due to the random nature of the experiments. Figure 5.6.C shows the detection rate for a larger circuit (`s38417`). Experiments on other circuits such as `s38584` show the same trend.

5.4.3 Different Selection Strategies

In order to analyze the effectiveness of the proposed algorithm another experiment was conducted in which the algorithm's strategy was modified to forcibly select all assertions first, and then traces; or alternatively, select all traces first and then the assertions. Figure 5.7 shows that the dynamic selection (concurrent selection of assertions and trace signals) leads to better detection rate when compared to the two modified versions.

In another experiment, the order in which the assertions, wires and flip-flops are selected were observed. The circuit `s5378` was provided to the algorithm. Wire budget for assertions was set to 64 while wire budget for trace selection was 48. After the execution, a total of 32 assertions were selected. If considered individually, those 32 assertions would use a total of 84 wires, but when considering wire-sharing among them they total to 64 wires, as enforced by the wire-budget for this particular resource. Figure 5.8 shows the order in which the selections were decided. The horizontal axis represents the decision sequence from left to right, and each marker represents the type of the selected resource at each step.

5.4.4 Time Distribution of the Suspects' List

As shown in Figure 5.9 the time distribution of the flip-flops in suspects' list can be within a relatively narrow window around the injection time. In this figure the injection time is assumed to be at time T=0. The green bars represent the distribution of the suspect flip-flops when only the information inside the trace buffer is used for root cause analysis. The blue line, on the other hand, shows the distribution when in addition to the trace buffer, the violation of assertion-checkers are also integrated

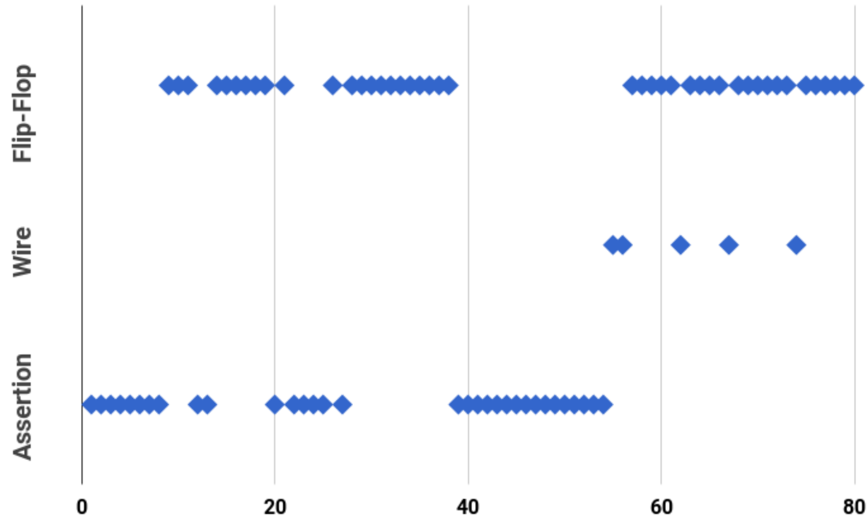


Figure 5.8: The order of decisions to select Assertions, wires from assertions, and flip-flops for circuit `s5378`. Wire budget for assertions is 64 and wire budget for trace selection is 48.

in the analysis. The added value of the assertion checkers is obvious in two respects: (1) more bit-flips are detected, and (2) the distribution has a higher peak around the injection time. This means that a number of smaller cores of UNSAT or equivalently several "shorter lists of suspects" are produced due to the violation of assertions.

Another visible difference between the time distribution of trace-only versus trace-assertion-combined experiments is the right skewness in the time distribution graphs of the combined method, as shown by the blue lines in Figure 5.9. This is due to the nature of the assertions and the depth of assertions in the assertion dataset. It can be deduced that the bit-flips can corrupt the state of the circuit in such a way that the assertions fail after one or more clock cycles. Each failed assertion, in turn has a depth, which corresponds to the number of samples it keeps for checking its particular property. A longer depth of an assertion, or more latency of an assertion

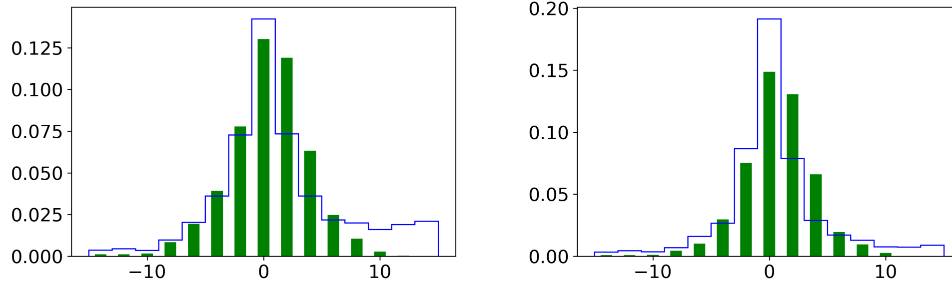


Figure 5.9: Time distribution of the flip-flops in the list of suspects for `s38417` assuming that the injection time is at time T=0. Green bars represent the distribution for when only trace information is used. Blue line illustrates the same distribution for when both violated assertions and trace signals are used for post analysis. Left: Trace width of 64 bits. Right: Trace width of 256 bits.

to fire results in a logical inconsistency among nets with higher timestamps, hence a longer tail in the time distribution histogram.

5.4.5 Runtime

As can be seen in algorithm 8, there are two types of loops: the outer `while` loop and the two inner `for` loops. The outer `while` loop runs roughly the same number of times as the sum of the given wire budget (W) and trace budget (T). Note that at every round of execution either a single trace signal is selected or one or multiple wires (for a single assertion-checker). Thus the runtime grows linearly to the sum of budgets for both assertion-wires and traced flip-flops. The two inner loops also cause a linear growth in runtime. The first `for` loops runs as many times as the number of assertions in the assertions data set. The second `for` loop scans all the flip-flops and also the wires associated with the assertions selected so far. This depends mainly on the size of the netlist. In summary the runtime grows linearly to all of these factors:

- number of trace signals to be selected (T)
- number of wires/flip-flops to be used for assertions (W)
- number of assertion in the dataset ($|\mathbb{C}_A|$)
- number of flip-flops in a netlist ($|\mathbb{C}_F|$)

Runtime of the algorithm (developed in C++) for `s5378` (64b of trace, 64 wires out of a total of 33,000 assertions) was 6.7 seconds. Runtimes for `s38417` (64b of trace, a total of 598 assertions) when assertion wires vary from 64 to 96 and 128, are respectively 535, 556, 571 seconds. Runtime is measured on Intel(R) core i7-2600 CPU.

5.5 Summary

In this chapter the following topics were discussed :

- Online assertion checkers were originally inherited from software design to hardware verification. They have the potential to be synthesized to hardware, integrated into the silicon die and be used for post-silicon validation. They can detect bit-flips as well as other deviations from specification.
- When dealing with electrical bugs that cause bit-flips in the logic domain, a violation of a hardware assertion can provide information in two ways:
    1. event detection (e.g. triggering another process such as data acquisition)
    2. provide the logic value of circuit nets that caused the violation.

- Together with on-chip trace buffers, assertions can provide useful information into the root-cause of a design error.

- An algorithm was proposed that integrates the knowledge from the assertions and trace buffers and **co-selects** a list of assertions and useful trace signals, concurrently. The method proposed in the previous chapter selected traced signals without taking any knowledge on assertions into account.

- The goal of the algorithm is to take advantage of trace buffers for some bit-flips and use assertions for others while saving cost of shared wires and hardware area.

- Results support that the extra cost of hardware assertions boost the bit-flip detection rate considerably.

Chapter 6

Conclusion and Future Work

In this thesis, it was shown that analyzing post-silicon traces can aid with the detection of bit-flips and root-cause analysis. In chapter 3, we presented a method that facilitates post-silicon validation by automatically detecting bit-flips using trace buffers. The behaviour of a digital circuit can be translated into a set of Boolean conditions. In this method, the Boolean representation is compared against the bits recorded in a trace buffer. In other words, the collected bits enforce a set of constraints for the Boolean representation of the circuit. The possible inconsistency between the circuit and the trace may be detected by a Boolean SAT solver. A SAT solver attempts to find a solution which is simultaneously compatible with both the circuit and the recorded trace. In the event that the two sets of conditions are incompatible, the SAT solver finds the conflicting subset of the problem. This subset will be used as a clue for root-cause analysis. It was shown that a list of suspects can further be provided by back-ward translation of the "Boolean problem subset" to "circuit nets".

In chapter 4, a selection algorithm was proposed that chooses the traced signals that lead to higher detection rate of bit-flips. Tracing a signal (flip-flop or wire within

the design) means that its Boolean state is recorded on the trace buffer at every clock cycle or in other words its value is known for the entire window of the trace history. Knowledge of several signals helps with reconstructing the state of others as well detecting inconsistencies, if any. Using a probability model, the knowledge of several signals and its effect on others non-traced signals is quantified and measured. This model is used at the core of the proposed algorithm to make decisions on which signals to trace.

It was shown that the set of signals selected by the algorithm can consistently lead to 10 to 15 percent more bit-flip detections when compared to a random selection. It was also shown that restoration-based methods that choose trace signals for higher state restoration (commonly used for functional bugs) can perform worse than random tracing, which already proves that they are not effective for root-causing electrical bugs. For example, as shown in chapter 4, by tracing 5 to 12% of the signals of a benchmark circuit, we can detect 15 to 25% of the bit-flips. During post-silicon debug it is common to run multiple sessions before a bit-flip is detected. Assuming that a flip-flop is considered "*covered*" if it is detected at least once out of 10 times, we can calculate a *coverage rate* for each selected set of traced signals. It was shown that the coverage rate of the same benchmark circuit (with 5 to 12% of all signals traced) can lead to 20% to 40% flip-flop coverage.

And finally in chapter 5 we presented another selection algorithm that concurrently selects traced signals and hardware assertions, with the objective of covering as many bit-flips as possible, while minimizing the estimated hardware cost. The hardware cost consists of the area of the assertion circuitry and the on-chip memory

used as a trace buffer. The same probability-based metric as in chapter 4 was employed and extended to integrate the knowledge from assertion checkers. It was shown that while spending the hardware budget (area) for a wider trace memory leads to higher detection rate, one can spend on both assertion checkers and trace buffers to bring in the best of both worlds. The newly added assertion checkers can lead to a significant boost both in detection rate and shrinkage of the list of suspects.

It was shown that the *"mixed co-selection"* strategy outperforms other selection strategies. More precisely, one can select all the required assertions first, and then all the traced signals, or on another extreme select all the traces before all the assertions. Our proposed algorithm, on the contrary, selects the traces and assertions concurrently while propagating the knowledge of each intermediate step towards the next decision-making step. As a result, in each iteration of the algorithm a traced wire, traced flip-flop or an assertion may be selected, depending on the current state of selections and the pool of possibilities.

In conclusion, the automatic bit-flip detection methods proposed in this dissertation have the potential to provide improved debugging during post-silicon validation. In pre-silicon, once the netlist is ready the traced signals can be selected automatically by the proposed algorithm. If a pool of assertions is also available, they can be filtered using the co-selection algorithm to cover as many bit-flips as the hardware budget allows. In post-silicon, the content of the trace buffer needs to be extracted and passed to the rest of proposed flow to detect the bit-flips and generate a list of suspect nets, where the bit-flips have occurred. This methodology is generic and can be applied to any digital block regardless of its application.

Possible Future Paths

Probably the single major point where one can improve the proposed methodology is on the selection algorithm. The algorithms in this work are greedy heuristics where each decision is the locally *best* solution. One can look for groups of signals or clusters within a netlist and by analyzing the structure, potentially make better decisions. Such an algorithm might select more than a signal at a time, for example several signals within a particular cluster or interconnected region of the circuit.

It is also worth considering to implement or emulate a time multiplexed trace buffer, at least for research purposes or as a feasibility study, in which a trace buffer is shared between multiple blocks. This can be beneficial in many aspects. Apart from the feasibility study, this can provide a more realistic signal probability (odds of a signal being logic 1) in an accelerated rate, much faster than what can be simulated. Currently almost all the experiments in this thesis are based on a ($P_1 = P_0 = 50\%$) assumption for all the traced signals. The signal probability from emulation can be fed to the selection algorithm to further refine the selection.

The last contribution of this thesis depends on a pool of assertions. This pool can potentially contain many assertions, including those that are of relatively high-quality. In this context a high-quality assertion is one that gets violated often, for one or more particular bit-flips. On the contrary a low-quality assertion is one that fires rarely or never, when a bit-flip occurs anywhere in the circuit. High quality assertions are favorable since they can provide useful information during a debug session. The quality of the proposed selection algorithm ultimately depends on the quality of the assertions provided to it. If a large number of high-quality assertions can be generated, for example using an automatic assertion generation algorithm,

then the chances of having a set of assertions and traced signals that catch more bit-flips gets higher.

All the suggestions for future work provided above can further aid the contributions from this thesis in improving the productivity and effectiveness of post-silicon validation and debugging. While no single approach will suffice in solving the major challenges faced in practice, it is anticipated that the algorithmic methods presented in this thesis will facilitate a step forward toward systematizing post-silicon validation and providing a stronger scientific foundation to this field.

Bibliography

[1] Z. Navabi, *VHDL: Analysis and Modeling of Digital Systems.* McGraw-Hill, Inc., 2nd ed., 1997.

[2] G. Gerosa, S. Curtis, M. D'Addeo, B. Jiang, B. Kuttanna, F. Merchant, B. Patel, M. H. Taufique, and H. Samarchi, "A sub-2 W low power ia processor for mobile internet devices in 45 nm high-k metal gate CMOS," *IEEE Journal of Solid-State Circuits*, vol. 44, pp. 73–82, Jan 2009.

[3] N. Weste and D. Harirs, *CMOS VLSI Design: A Circuits and Systems Perspective.* Pearson, 4th ed., 2010.

[4] "Dictionary by Merriam-Webster." `https://www.merriam-webster.com/dictionary/computer`, 2017.

[5] "Meanings and definitions of words at dictioinary.com." `http://www.dictionary.com/browse/computer`, 2017.

[6] C. Hope, "When was the first computer invented." `https://www.computerhope.com/issues/ch000984.htm`, 2017.

[7] M. Kanellos, "Intel's accidental revolution." https://archive.is/20120711020441/http://news.cnet.com/Intels-accidental-revolution/2009-1001_3-275806.html, 2001.

[8] G. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. 38, no. 8, 1965.

[9] L. Durant, O. Giroux, M. Harris, and N. Stam, "Inside Volta: The Worlds Most Advanced Data Center GPU." http://archive.is/d698Y.

[10] Intel, "The Story of the Intel(R) 4004." http://archive.is/zdHw4.

[11] H. Grtker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Springer, 2002.

[12] Z. Navabi, *Verilog Digital System Design*. McGraw Hill, 2nd ed., 2005.

[13] C. Y. Chang and S. M. Sze., *ULSI Technology*. Mcgraw-Hill College Publications, 1996.

[14] M. Ganai and A. Gupta, *SAT-Based Scalable Formal Verification Solutions*. Springer, 5 ed., 2007.

[15] P. Rashinkar, P. Paterson, and L. Singh, *System-on-a-Chip Verification*. Springer, 2013.

[16] S. Devadas, A. Ghosh, and K. Keutzer, "An observability-based code coverage metric for functional simulation," in *Proceedings of the IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pp. 418–425, IEEE Computer Society, 1996.

[17] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv, "User defined coverage - a tool supported methodology for design verification," in *Proceedings IEEE/ACM Design Automation Conference (DAC)*, pp. 158–163, ACM, 1998.

[18] D. Moundanos, J. A. Abraham, and Y. V. Heskote, "A unified framework for design validation and manufacturing test," in *IEEE International Test Conference (ITC)*, pp. 875–884, Oct 1996.

[19] H. Foster, A. Krolnik, and D. Lacey., *Assertion-Based Design.* Springer, 2nd ed., 2005.

[20] Q. Wang, R. Kassa, W. Shen, N. Ijih, B. Chitlur, M. Konow, D. Liu, A. Sheiman, and P. Gupta, "An FPGA based hybrid processor emulation platform," in *International Conference on Field Programmable Logic and Applications*, pp. 25–30, Aug 2010.

[21] D. Anastasakis, R. Damiano, H. K. T. Ma, and T. Stanion, "A practical and efficient method for compare-point matching," in *Proceedings IEEE/ACM Design Automation Conference (DAC)*, pp. 305–310, 2002.

[22] D. Perry and H. Foster, *Applied Formal Verification.* McGraw-Hill, 2005.

[23] S. Sutherland, P. Moorby, S. Davidmann, and P. Flake., *SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling.* Springer, 2nd ed., 2006.

[24] "The 2012 Wilson Research Group Functional Verification Study." `http://blogs.mentor.com/verificationhorizons/blog/2013/04/23/`

`prologue-the-2012-wilson-research-group-functional-verification-study/`, 2012.

[25] J. Goodenough and R. Aitken, "Post-silicon is too late avoiding the 50 million paperweight starts with validated designs," in *IEEE/ACM Design Automation Conference (DAC)*, pp. 8–11, June 2010.

[26] L. Wang, C. Wu, , and X. Wen, *VLSI Test Principles and Architectures: Design for Testability*. Morgan Kaufmann Publications, 1st ed., 2006.

[27] P. H. Bardell, W. H. McAnney, and J. Savir, *Built-in Test for VLSI: Pseudorandom Techniques*. Wiley-Interscience publication. Wiley, 2nd ed., 1987.

[28] B. Koenemann and S. Pateras, "Built-in self-test (BIST) in the era of deep submicron technology," in *IEEE Technical Applications Conference. Northcon/96. Conference Record*, pp. 312–315, Nov 1996.

[29] V. Gherman, H. J. Wunderlich, H. Vranken, F. Hapke, M. Wittke, and M. Garbers, "Efficient pattern mapping for deterministic logic BIST," in *International Conferce on Test*, pp. 48–56, Oct 2004.

[30] N. A. Touba and E. J. McCluskey, "Bit-fixing in pseudorandom sequences for scan BIST," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, pp. 545–555, Apr 2001.

[31] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design. Information Technology: Transmission, Processing and Storage*. Springer, 2004.

[32] P. Yeung and K. Larsen, "Practical assertion-based formal verification for soc designs," in *International Symposium on System-on-Chip*, pp. 58–61, Nov 2005.

[33] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rulke, "Advanced verification by automatic property generation," *IET Computers Digital Techniques*, vol. 3, pp. 338–353, July 2009.

[34] M. Boule, J. S. Chenard, and Z. Zilic, "Assertion checkers in verification, silicon debug and in-field diagnosis," in *8th International Symposium on Quality Electronic Design (ISQED)*, pp. 613–620, March 2007.

[35] S. Mitra, S. A. Seshia, and N. Nicolici, "Post-silicon validation opportunities, challenges and recent advances," in *IEEE/ACM Design Automation Conference (DAC)*, pp. 12–17, June 2010.

[36] "International Test Conference (ITC)." `http://www.itctestweek.org/`, 2017.

[37] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer, 2002.

[38] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A reconfigurable design-for-debug infrastructure for SoCs," in *IEEE/ACM Design Automation Conference (DAC)*, pp. 7–12, 2006.

[39] E. Anis and N. Nicolici, "On using lossless compression of debug data in embedded logic analysis," in *IEEE International Test Conference (ITC)*, pp. 1–10, Oct 2007.

[40] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*. IEEE Press, Piscataway, NJ, 2nd ed., 1994.

[41] E. J. McCluskey, *Logic Design Principles: With Emphasis on Testable Semiconductor Circuits*. Prentice Hall, Englewood Cliffs, NJ, 1986.

[42] A. Nahir, A. Ziv, M. Abramovici, A. Camilleri, R. Galivanche, B. Bentley, H. Foster, A. Hu, V. Bertacco, and S. Kapoor, "Bridging pre-silicon verification and post-silicon validation," in *IEEE/ACM Design Automation Conference (DAC)*, pp. 94–95, June 2010.

[43] N. Nicolici, "On-chip stimuli generation for post-silicon validation," in *2012 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pp. 108–109, Nov 2012.

[44] S. K. Sadasivam, S. Alapati, and V. Mallikarjunan, "Test generation approach for post-silicon validation of high end microprocessor," in *2012 15th Euromicro Conference on Digital System Design*, pp. 830–836, Sept 2012.

[45] A. Adir, S. Copty, S. Landa, A. Nahir, G. Shurek, A. Ziv, C. Meissner, and J. Schumann, "A unified methodology for pre-silicon verification and post-silicon validation," in *IEEE/ACM Design, Automation & Test in Europe (DATE)*, pp. 1–6, March 2011.

[46] X. Shi and N. Nicolici, "On-chip cube-based constrained-random stimuli generation for post-silicon validation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, pp. 1012–1025, June 2016.

[47] X. Shi and N. Nicolici, "Generating cyclic-random sequences in a constrained space for in-system validation," *IEEE Transactions on Computers*, vol. 65, pp. 3676–3686, Dec 2016.

[48] A. B. Kinsman, H. F. Ko, and N. Nicolici, "In-system constrained-random stimuli generation for post-silicon validation," in *IEEE International Test Conference (ITC)*, pp. 1–10, Nov 2012.

[49] X. Shi, *Constrained-Random Stimuli Generation for Post-Silicon Validation*. PhD dissertation, McMaster University, 2016.

[50] B. Vermeulen, T. Waayers, and S. K. Goel, "Core-based scan architecture for silicon debug," in *IEEE International Test Conference (ITC)*, pp. 638–647, 2002.

[51] S. Tang and Q. Xu, "In-band cross-trigger event transmission for transaction-based debug," in *Design, Automation and Test in Europe (DATE)*, pp. 414–419, March 2008.

[52] H. F. Ko and N. Nicolici, "Resource-efficient programmable trigger units for post-silicon validation," in *IEEE European Test Symposium (ETS)*, pp. 17–22, May 2009.

[53] T. Hong, Y. Li, S. B. Park, D. Mui, D. Lin, Z. A. Kaleq, N. Hakim, H. Naeimi, D. S. Gardner, and S. Mitra, "Qed: Quick error detection tests for effective post-silicon validation," in *IEEE International Test Conference (ITC)*, pp. 1–10, Nov 2010.

[54] D. Lin, T. Hong, Y. Li, E. S, S. Kumar, F. Fallah, N. Hakim, D. S. Gardner, and S. Mitra, "Effective post-silicon validation of system-on-chips using quick error detection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, pp. 1573–1590, Oct 2014.

[55] H. Sohofi and Z. Navabi, "Assertion-based verification for system-level designs," in *Fifteenth International Symposium on Quality Electronic Design*, pp. 582–588, March 2014.

[56] R. Sebastian, S. R. Mary, G. M, and A. Thomas, "Assertion based verification of sgmii ip core incorporating axi transaction verification model," in *International Conference on Control Communication Computing India (ICCC)*, pp. 585–588, Nov 2015.

[57] I. Syafalni, N. Surantha, D. K. Lam, N. Sutisna, Y. Nagao, K. Wakasugi, Y. Tongxin, H. Ochi, and T. Tsuchiya, "Assertion-based verification of industrial wlan system," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 982–985, May 2016.

[58] Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal, *FoCs – Automatic Generation of Simulation Checkers from Formal Specifications*, pp. 538–542. Springer Berlin Heidelberg, 2000.

[59] M. Boule and Z. Zilic, *Generating Hardware Assertion Checkers.* Springer, 1st ed., 2008.

[60] M. Pellauer, M. Lis, D. Baltus, and R. Nikhil, "Synthesis of synchronous assertions with guarded atomic actions," in *Proceedings ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, pp. 15–24, July 2005.

[61] M. Boule and Z. Zilic, "Efficient automata-based assertion-checker synthesis of psl properties," in *IEEE International High Level Design Validation and Test Workshop*, pp. 69–76, Nov 2006.

[62] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rulke, "Automatic generation of complex properties for hardware designs," in *Design, Automation and Test in Europe (DATE)*, pp. 545–548, March 2008.

[63] S. Hertz, D. Sheridan, and S. Vasudevan, "Mining hardware assertions with guidance from static analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, pp. 952–965, June 2013.

[64] A. Hekmatpour and A. Salehi, "Block-based schema-driven assertion generation for functional verification," in *Asian Test Symposium (ATS)*, pp. 34–39, Dec 2005.

[65] H. F. Ko and N. Nicolici, "Automated trace signals identification and state restoration for improving observability in post-silicon validation," *IEEE/ACM Design, Automation & Test in Europe (DATE)*, pp. 1298–1303, Mar. 2008.

[66] X. Liu and Q. Xu, "Trace signal selection for visibility enhancement in post-silicon validation," *IEEE/ACM Design, Automation & Test in Europe (DATE)*, pp. 1338–1343, Apr. 2009.

[67] S. Prabhakar and M. Hsiao, "Using Non-trivial Logic Implications for Trace Buffer-Based Silicon Debug," in *Asian Test Symposium (ATS)*, pp. 131–136, IEEE Computer Society, Nov. 2009.

[68] H. Shojaei and A. Davoodi, "Trace signal selection to enhance timing and logic visibility in post-silicon validation," in *IEEE/ACM International Conference on Computer-Aided Design, ICCAD*, pp. 168–172, Nov. 2010.

[69] K. Basu and P. Mishra, "Efficient trace signal selection for post silicon validation and debug," *IEEE International Conference on VLSI Design*, pp. 352–357, Jan 2011.

[70] X. Liu and Q. Xu, "On signal selection for visibility enhancement in trace-based post-silicon validation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, pp. 1263–1274, aug 2012.

[71] M. Li and A. Davoodi, "A Hybrid Approach for Fast and Accurate Trace Signal Selection for Post-Silicon Debug," *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 485–490, Jan. 2013.

[72] E. Hung and S. J. E. Wilton, "Scalable signal selection for post-silicon debug," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1103–1115, June 2013.

[73] K. Basu and P. Mishra, "RATS: Restoration-aware trace signal selection for post-silicon validation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 605–613, Apr. 2013.

[74] K. Rahmani, P. Mishra, and S. Ray, "Scalable trace signal selection using machine learning," in *IEEE 31st International Conference on Computer Design (ICCD)*, pp. 384–389, Oct. 2013.

[75] M. Li and A. Davoodi, "Multi-mode trace signal selection for post-silicon debug," *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 640–645, Jan. 2014.

[76] K. Rahmani, P. Mishra, and S. Ray, "Efficient trace signal selection using augmentation and ILP techniques," in *IEEE International Symposium on Quality Electronic Design (ISQED)*, pp. 148–155, 2014.

[77] S. Ma, D. Pal, R. Jiang, S. Ray, and S. Vasudevan, "Can ' t See the Forest for the Trees : State Restoration ' s Limitations in Post-silicon Trace Signal Selection," in *IEEE/ACM International Conference on Computer-Aided Design, ICCAD*, pp. 1–8, IEEE Press, Nov. 2015.

[78] K. Iwata, A. M. Gharehbaghi, M. B. Tahoori, and M. Fujita, "Post silicon debugging of electrical bugs using trace buffers," in *IEEE 26th Asian Test Symposium (ATS)*, pp. 189–194, Nov 2017.

[79] P. Patra, "On the cusp of a validation wall," *IEEE Design Test of Computers*, vol. 24, pp. 193–196, March 2007.

[80] S.-B. Park and S. Mitra, "IFRA: Instruction footprint recording and analysis for post-silicon bug localization in processors," in *IEEE/ACM Design Automation Conference (DAC)*, pp. 373–378, June 2008.

[81] S.-B. Park, A. Bracy, H. Wang, and S. Mitra, "BLoG: Post-Silicon Bug Localization in Processors using Bug Localization Graphs," in *IEEE/ACM Design Automation Conference (DAC)*, p. 368, 2010.

[82] A. Veneris, "Fault diagnosis and logic debugging using boolean satisfiability," in *4th International Workshop on Microprocessor Test and Verification*, pp. 60–65, May 2003.

[83] T. Larrabee, "Test pattern generation using Boolean satisfiability," *IEEE Transactions on CAD*, vol. 11, pp. 4–15, Jan 1992.

[84] Y. Vizel, G. Weissenbacher, and S. Malik, "Boolean Satisfiability Solvers and Their Applications in Model Checking," *Proceedings of the IEEE*, vol. 103, pp. 2021–2035, Nov. 2015.

[85] A. Sülflow, G. Fey, R. Bloem, and R. Drechsler, "Using unsatisfiable cores to debug multiple design errors," in *18th ACM Great Lakes Symposium on VLSI (GLSVLSI)*, pp. 77–82, 2008.

[86] C. S. Zhu, G. Weissenbacher, and S. Malik, "Silicon fault diagnosis using sequence interpolation with backbones," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 348–355, IEEE Press, 2014.

[87] A. Nahir, M. Dusanapudi, S. Kapoor, K. Reick, W. Roesner, K. Schubert, K. Sharp, and G. Wetli, "Post-silicon validation of the IBM POWER8 processor," in *IEEE/ACM Design Automation Conference (DAC)*, pp. 1–6, 2014.

[88] P. Taatizadeh and N. Nicolici, "Automated selection of assertions for bit-flip detection during post-silicon validation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 2118–2130, 2016.

[89] G. Tseytin, "On the complexity of derivation in propositional calculus.," in *Studies in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics* (A. Slisenko, ed.), pp. 234–259, 1968.

[90] "SAT competition 2009: Benchmark submission guidelines." `http://www.satcompetition.org/2009/format-benchmarks2009.html`, 2009-01-13.

[91] A. Biere, "Picosat essentials," *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, vol. 4, pp. 75–97, 2008.

[92] N. Een and N. Sorensson, "An extensible SAT-solver." `http://minisat.se/downloads/MiniSat.pdf`, 2003.

[93] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1929–1934 vol.3, May 1989.

[94] "ITC99 benchmark circuits collection." `http://www.cad.polito.it/downloads/tools/itc99.html`, April 2002.

[95] H. Ko and N. Nicolici, "Algorithms for state restoration and trace-signal selection for data acquisition in silicon debug," *IEEE Transactions on CAD*, vol. 28, pp. 285–297, Feb 2009.

[96] A. Vali and N. Nicolici, "Satisfiability-Based Analysis of Failing Traces during Post-silicon Debug," *IEEE 24th North Atlantic Test Workshop (NATW)*, pp. 17–22, 2015.

[97] C. Spear, *SystemVerilog for Verification, Second Edition: A Guide to Learning the Testbench Language Features*. Springer Publishing Company, Incorporated, 2nd ed., 2008.

[98] M. Boule, J.-S. Chenard, and Z. Zilic, "Assertion checkers in verification, silicon debug and in-field diagnosis," in *IEEE International Symposium on Quality Electronic Design (ISQED)*, pp. 613–620, March 2007.

[99] M. Gao and K.-T. Cheng, "A case study of time-multiplexed assertion checking for post-silicon debugging," in *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pp. 90–96, June 2010.

[100] S. Hertz, D. Sheridan, and S. Vasudevan, "Mining hardware assertions with guidance from static analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, pp. 952–965, June 2013.

[101] P. Taatizadeh and N. Nicolici, "A methodology for automated design of embedded bit-flips detectors in post-silicon validation," in *ACM/IEEE Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 73–78, March 2015.

[102] P. Taatizadeh, *On Using Hardware Assertion Checkers for Bit-flip Detection in Post-Silicon Validation*. PhD dissertation, McMaster University, 2017.