On Using Hardware Assertion Checkers for Bit-flip Detection in Post-Silicon Validation

ON USING HARDWARE ASSERTION CHECKERS FOR BIT-FLIP DETECTION IN POST-SILICON VALIDATION

BY

POUYA TAATIZADEH, B.Eng. 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

DOCTOR OF PHILOSOPHY

Doctor of Philosophy (2017)	McMaster University
(Electrical & Computer Engineering)	Hamilton, Ontario, Canada

TITLE:	On Using Hardware Assertion Checkers for Bit-flip Detection in Post-Silicon Validation
AUTHOR:	Pouya Taatizadeh B.Eng., (Electronic Engineering) University of Southampton, Southampton, UK M.Sc., (System-On-Chip) University of Southampton, Southampton, UK
SUPERVISOR:	Prof. Nicola Nicolici
NUMBER OF PAGES:	xviii, 184

To My Family

Abstract

With the rising demand for integrating more features in a single product, modern designs feature more and more functionality on a single die. The complexity resulted from this growth makes it more difficult to guarantee that all errors are detected and fixed before the product is manufactured. Efforts carried out before the design is manufactured, known as pre-silicon verification, are unable to detect all the errors. Electrical errors, such as those caused by cross-talk or power droops, are particularly difficult to catch during the pre-silicon phase because of the insufficient accuracy of device models, which is often traded-off against simulation time. This challenge is further aggravated by the rising number of voltage domains, especially if subtle errors are excited in unique electrical states. These electrically-induced subtle errors most commonly manifest in the logic domain as *bit-flips* in flip-flops. Therefore, once the design prototypes are available, the verification tasks continue on them to identify and eliminate errors that have escaped pre-silicon verification phase and have the potential to cause catastrophic problems if they remain undetected. This task is commonly referred to as *post-silicon validation* and has become an important step in the design flow of system-on-chip devices.

Limited internal node observability is one of the main challenges of post-silicon validation because it causes long error detection latencies. The existing approaches that try to improve this limited observability usually rely on *ad-hoc* methods which are difficult to maintain when particular changes are requested from one project to another. Hence in this thesis, we propose novel systematic methods which can be used for automatic generation of on-chip blocks that improve internal observability and reduce error detection latency in post-silicon validation.

First we propose an *automated methodology* that generates and selects hardware assertions for bit-flip detection in post-silicon validation. As part of this methodology, we will introduce two quantitative metrics, the bit-flip coverage estimate and the flip-flop coverage estimate that can be used to assess the quality of the selected assertions. Next, we improve the run-time and the accuracy of our proposed methodology by using *emulation platforms* that can automatically be integrated into the proposed flow to enable fast, yet accurate assertion selection. Finally in the last contribution, we will propose a novel solution for *automatic generation of hardware assertions* by leveraging the internal mechanism of Boolean satisfiability solvers, namely learned clauses. In addition, we will formally evaluate the potential of the discovered hardware assertions using an incremental SAT solving approach to assess these hardware assertions for bit-flip detection. We will show that by using this new method, the proportion of errors that can be detected are improved while the area overhead is reduced.

State-of-the-art post-silicon validation practices rely primarily on *ad-hoc* methods for bit-flip detection and there is a lack of systematic methods that can be applied to generic digital blocks. Consequently, the contributions from this thesis are a step towards introducing assertion-based methods and architectures that facilitate systematic bit-flip detection in post-silicon validation.

Acknowledgements

Similar to every big achievement in life, it is impossible to list everybody who had an impact on how I managed to close this chapter of my life. However there are those whose help and influence are unforgettable and hence deserve my appreciation.

I would like to thank my family who have always stayed by my side, supported me and have given me nothing but love and courage. Their presence is the most valuable thing I possess in life.

I would also like to express my deepest gratitude to my advisor, Prof. Nicola Nicolici for his endless patience and unswerving commitment which gave me the opportunity to grow both technically and personally. It goes without saying that this work would have been impossible without his tireless efforts and guidance. When I joined McMaster, I was "the youngest" in his own words. I am not sure if the implication of that statement holds true any more.

I also would like to thank my PhD committee members, Dr. Shahram Shirani and Dr. Yaser Haddara for their precious time in reviewing my thesis and their valuable advice to improve my work.

I am grateful to both the former and present members of the Computer-aided Design and Test laboratory members at McMaster University, Dr. Adam Kinsman, Dr. Henry Ko, Dr. Zahra Lak, Dr. Jason Thong, Dr. Xiaobing Shi, Phil Kinsman and Amin Vali for their valuable assistance and helpful discussions that enriched the paths that I followed in my research. I would also like to thank Yasamin Fazliani for her support that significantly lessened the burden of this whole journey.

Finally, I would like to thank the administrative and technical staff of the ECE department at McMaster for their assistance.

Notation and abbreviations

ABV	Assertion Based Verification
ASIC	Application Specific Integrated Circuits
ATE	Automatic Test Equipment
ATPG	Automatic Test Pattern Generation
BDD	Boolean Decision Diagram
CNF	Conjunctive Normal Form
CPU	Central Processing Unit
CUT	Circuit Under Test
CUV	Circuit Under Validation
DFD	Design for Debug
DFT	Design For Test
DUT	Design Under Test
EDA	Electronic Design Automation
ELA	Embedded Logic Analyzer
FPGA	Field Programmable Gate Array
FSM	Finite-State Machine
IC	Integrated Circuit

IP	Intellectual Property
LUT	Look Up Table
MOS	Metal Oxide Semiconductor
OVL	Open VErification Library
PSL	Property Specification Language
PTE	Programmable Trigger Units
QBF	Quantified Boolean Formula
QED	Quick Error Detection
SFF	Scan Flip-flop
SoC	System-on-a-chip
SVA	System Verilog Assertion
TAP	Test Access Port
TDI	Test Data Input
VCD	Value Change Dump
VLSI	Very Large Scale Integrated Circuits

Contents

List of Figures

Chapter 1

Introduction

Recent advancements in process technologies have enabled very large scale integrated (VLSI) circuits to be built with millions and billions of transistors. As the digital integrated circuits (ICs) become more complex, ensuring their correct functional behavior also become more challenging. Nevertheless, the existing verification techniques that are used prior to tape-out (pre-silicon phase) are inadequate to detect and eliminate all the errors (bugs) that have the potential to cause catastrophic problems if they remain undetected. Hence, once the early chip prototypes are available and before committing to high-volume manufacturing, it is sensible to continue the verification process on those prototypes to detect the errors that have escaped the pre-silicon phase. This task is commonly referred to as *Post-Silicon Validation*.

Because of the sheer complexity of the designs, validation of VLSI circuits cannot be done manually which justifies the need for designing sophisticated automated methodologies. This motivates the contributions described in this thesis which focuses on introducing new concepts and automated methodologies for improving the existing techniques in post-silicon validation.

The rest of this chapter is organized as follows: At first, a quick overview of how computing has evolved and how computer-aided design has stepped into the chip design cycle is provided. Afterwards, important phases of the chip design flow are explained which determines the position of post-silicon validation in this flow. Finally, the chapter is closed by outlining the structure of the thesis.

1.1 History of Computing

The introduction of computational engines dates back to the 19th century [10]. In 1832, Babbage designed a mechanical calculator called *Difference Engine* that removed sources of mistakes in mathematical computations. The complexity and cost of the design made the concept impractical. Later on, the electrical solutions that were based on magnetically controlled switches (or relays) turned out to be more cost effective. Nonetheless, the age of digital electronic computing only started in full with the introduction of the vacuum tubes. While vacuum tubes were initially being used exclusively for analog applications, it was realized that they are also useful for digital computations. ENIAC (designed for computing artillery firing tables) and the UNIVAC (the first commercial computer) [11] are examples of the systems built based on this technology. High integration density soon pushed this design technology to its limits. In addition, reliability problems together with excessive power consumptions made manufacturing larger machines practically infeasible.

The invention of transistors at Bell Telephone Laboratories [12] together with the introduction of bipolar transistors by Schockley [13] in 1949 revolutionized this trend. It however took some time until these new technologies translated into a set of commercial integrated-circuit incorporating logic gates, called the Fairchild Micrologics family [14]. The theories and principles of MOSFET transistors had been investigated by J. Lilienfeld in 1925 and O. Heil in England in 1935 [10]. However, their practicality had not been realized due to insufficient knowledge about the behavior of these materials. Once this roadblock was resolved, MOS digital ICs started to appear in 1970s [10] which revolutionized the design of the computers.

Complexity and integration density of digital ICs have been continuously rising in the past couple of decades. Gordon Moore, the co-founder of Intel predicted in 1960s that the number of transistors on a single die would grow exponentially [15], a prediction that later on proved to be true as shown in Figure 1.1.

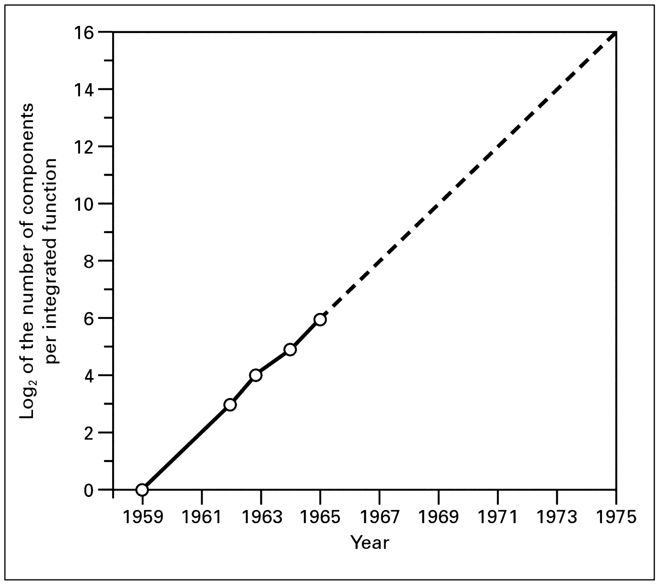


Figure 1.1: Number of components on an integrated circuit available in [1].

Clearly, as the integration density increases and designs become more complex, the existing design and verification methodologies need to evolve to keep pace with this trend. As stated earlier, this complexity together with aggressive time-to-market pressure necessitates the engineers to rely on automated, yet efficient methodologies that enables them to meet these constraints while keeping up with the new requirements that improve the performance of digital ICs.

Computer-Aided Design (CAD) helps engineers to rely on computers during different phases of IC development. Nonetheless, with the continuing rise in the complexity of digital ICs, demands for extended capability of CAD tools in different phases of IC design flow increases [16]. Therefore, there has been a tremendous amount of research done in the past five decades that addresses different stages of IC design flow with main emphasis on scalable tool development [17]. In the following subsections, the different stages of IC development are explained together with their unique challenges.

1.2 Design Flow of VLSI Circuits

There are three major steps that comprise the design of VLSI circuits: *Specification*, *Implementation* and *Manufacturing* as shown in Figure 1.2. The specification determines the expected functionality of the VLSI circuits and is typically expressed in high-level description languages such as SystemC [18]. Details of the internal components such as CPUs, memories, internal bus, etc., are abstracted from the specification. Therefore, specification is concerned with computations and data transfers without elaborating into specific details. To detail more information about the implementation, hardware description languages (HDLs) such as VHDL [16], Verilog [19] and System Verilog [20] are commonly used. When using HDLs, the flow of data between the internal registers as well as any logical operation are described at the register transfer level (RTL) abstraction level. By using the RTL model of the design, *synthesis* can be performed. Synthesis is defined as the process of transforming the circuit model from a higher abstraction level to a lower one. During synthesis, the transfer functions are optimized and transformed into logic equations that can be mapped to the gate-level components available from the target technology.

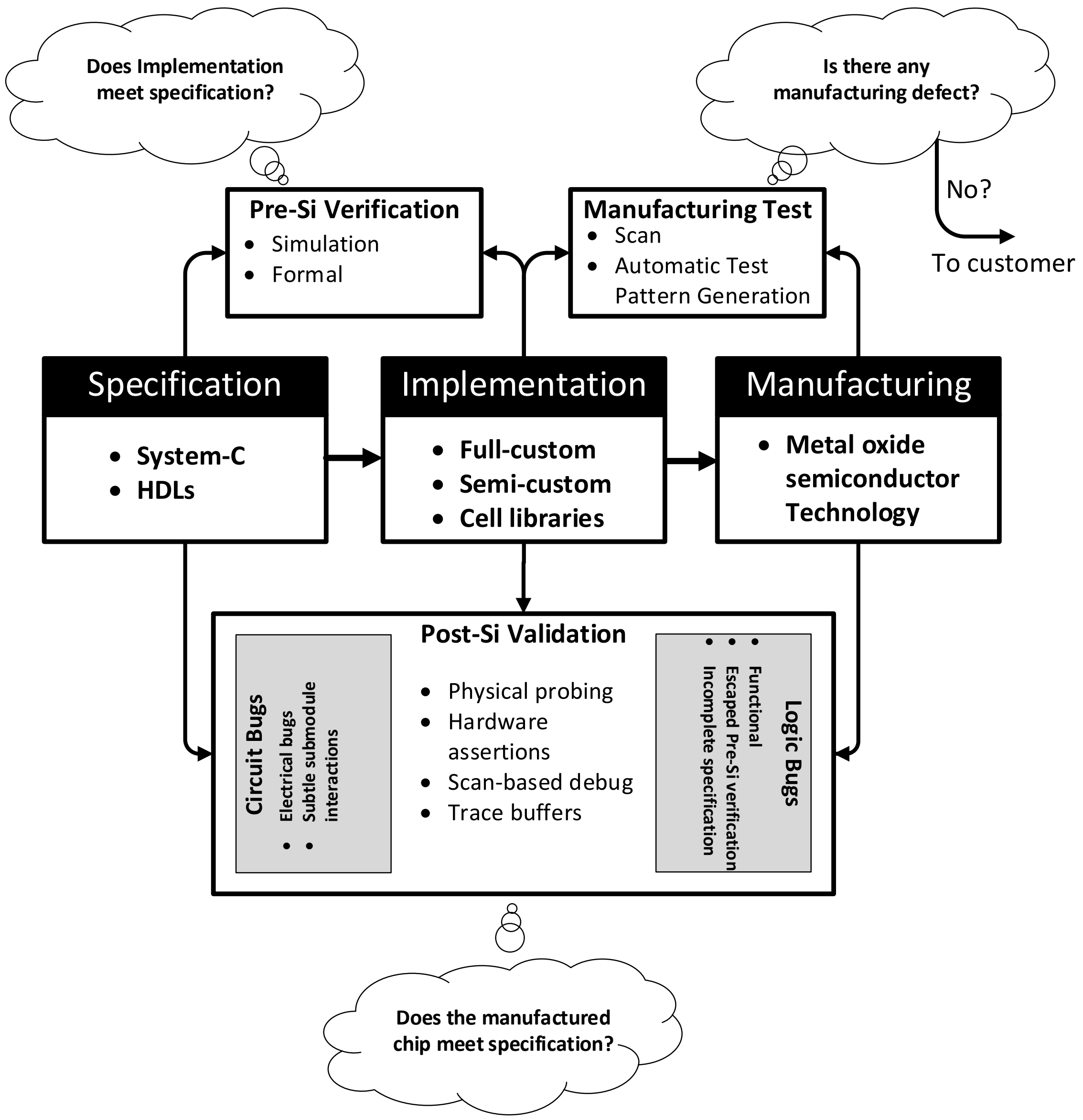


Figure 1.2: Typical chip design flow representing different phases of the design cycle.

1.3 Pre-Silicon Verification

Following to preparing the RTL model of the design from the specification, many verification steps are carried out before design's tape-out to ensure that the implemented design meets the given specification. This important step is known as pre-silicon verification. With the rising trend in complexity of digital hardware, it has been reported that as much as 70% of the product development cycle is spend at this time-consuming step [21]. There are two types of pre-silicon verification techniques that are commonly used: *simulation-based* approach and *formal verification* approach [22].

Event-based simulation techniques using functional simulators is a popular method that models the dynamic behavior of the simulated circuits by loading the events one by one and propagating them through the design. However, one key limitation of all simulation-based techniques is the inability to guarantee that the design is 100% error free. This is because with the large number of inputs in today's designs, there exist a large number of input enumerations that need to be applied to the circuit in order to ensure that the design is error-free [23]. Therefore, it is sensible to prepare testbenches that apply constrained-random stimuli to the circuit. The effectiveness of these tests can be evaluated using several coverage metrics such as code coverage, tag coverage [24], event coverage [25] and state-machine coverage [26]. In addition to this, some monitors known as assertions are added to the design to check specific properties that have to hold true indefinitely [27]. These assertions normally capture high-level design properties that should be respected by the design, for instance a request and acknowledgment relation in a data transfer protocol.

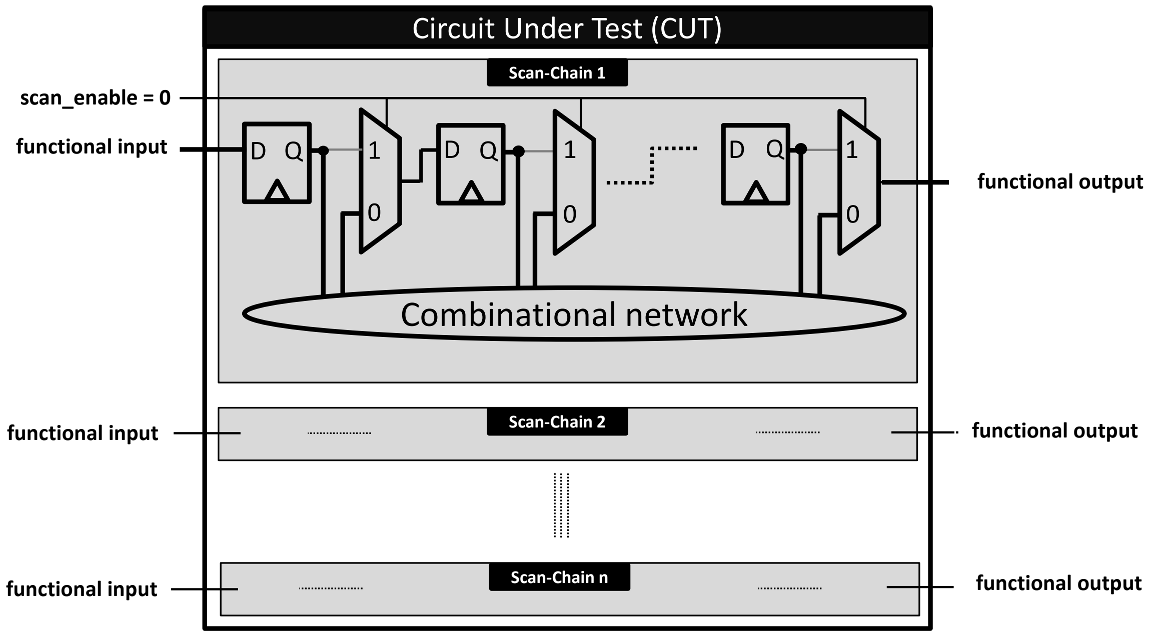
Formal verification on the other hand is concerned with mathematically proving or disproving the correctness of a system with respect to certain specifications or properties [21]. Thus, formal verification performs an exhaustive exploration of all possible behavior rather than an explicit enumeration of the possible behaviors as in functional simulation-based approaches [28]. Formal verification methods can broadly be classified into two categories: *formal model checking* and *formal equivalence checking* [29]. The former uses mathematical techniques to verify behavioral properties of a design and the latter uses mathematical techniques to prove equivalence of the design under verification (DUV) to a reference design. With the rising interest in performing formal verification, techniques for automating this process have evolved [30]. However one should note that although formal verification can ensure complete design correctness with respect to the given design properties without needing a testbench, the design is verified against the specification from which the circuit model is extracted. Therefore, if the specification is incomplete or has flaws, then the correctness of the formal verification is compromised.

As mentioned before, although simulation is a must for functional verification that helps with detecting the majority of bugs, it cannot provide 100% coverage of the entire design's state-space and hence, cannot ensure 100% design correctness. This justifies the need for an extra step in the chip-design flow, called *post-silicon validation* that is introduced in section 1.5 and detailed in chapter 2, to further eliminate design errors before high-volume manufacturing. The contributions from this thesis are placed within the context of post-silicon validation.

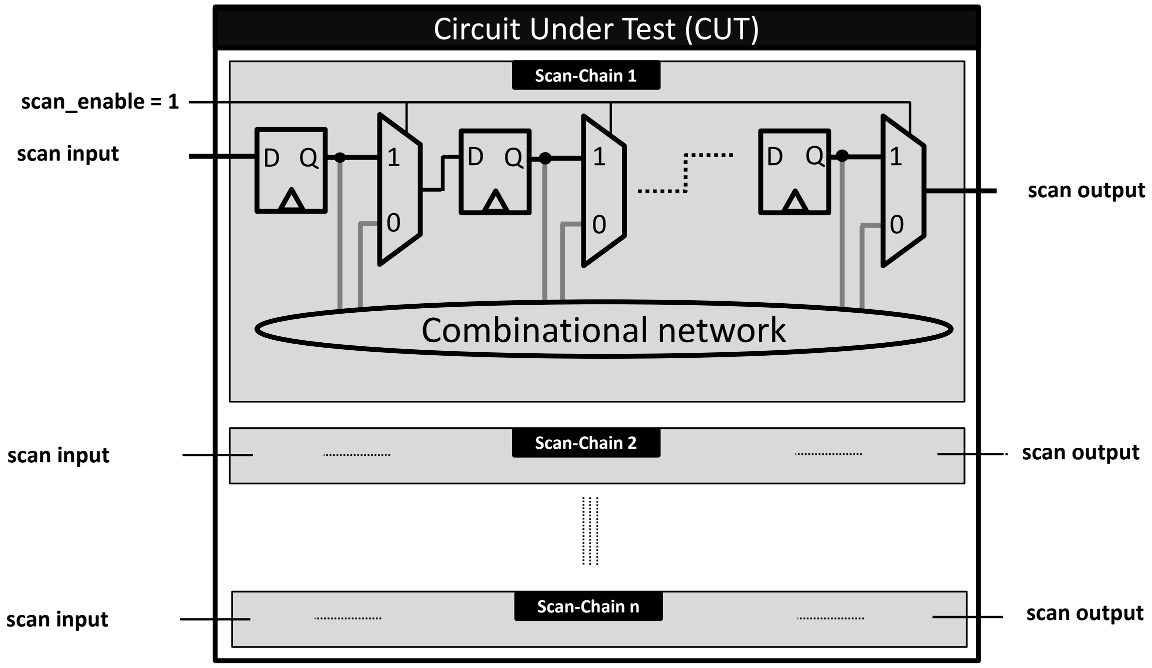
1.4 Manufacturing Test

Manufacturing test is an important phase in chip design flow that checks the manufactured circuits against the implementation to detect *physical defects* that can cause the circuit to malfunction [31]. During manufacturing test, different input vectors are applied to the circuit and the corresponding output responses are checked for pass/fail analysis. The process of applying the set of input vectors to the circuit under test (CUT) and the corresponding check against the golden model is normally done by an automatic test equipment (ATE). There exists two types of tests for which input vectors and corresponding responses are produced: *functional test* and *structural test.* During functional tests, the entire functionality of the circuit is tested. For instance, an exhaustive functional test of a 64-bit ripple carry adder requires 2^{129} number of test vectors. Using an ATE that works at 1GHz, the complete test requires 2.158×10^{22} years to finish [31]. Therefore, exhaustive functional test is impractical. On the other hand, structural testing relies on the netlist structure of the circuit. Different fault models are defined to guide the automatic test pattern generation (ATPG) algorithms to be developed for test generation, application and evaluation. Some common fault-models are single stuck-at, path-delay and bridging fault models. The most commonly used one is the single-stuck at model where it assumes a logic net to be stuck at logic value 0 (s-a-0) or stuck at logic value 1 (s-a-1) [32]. To compare this with functional testing, when using the single-stuck at fault model for the 64-bit ripple-carry adder, only 1728 stuck-at faults would need to be evaluated with 1728 test patterns in the worst-case [31].

For modern large and complex VLSI designs, it is very common to have some internal state signals that cannot be easily controlled by primary inputs. Therefore,



(a) Functional mode.



(b) Scan mode.

Figure 1.3: Scan-chain architecture for manufacturing test

to improve controllability and observability of the internal nodes, design for testability (DFT) circuitry is employed. The most commonly used DFT methodology is the scan-based DFT where all or part of the internal state-elements (i.e., flip-flops) are replaced with scan flip-flops (SFFs). A SFF is a flip-flop which has a multiplexer at its input. During normal operation, the enable signal of the multiplexer is at logic 0 and the data flow is done normally as shown in Figure 1.3a. On the other hand, during the test mode, the scan enable will be set to logic 1 and the SFFs form a shift register structure known as a scan-chain, as shown in Figure 1.3b. Input test vectors are serially shifted into the scan chain and the circuit response is analyzed at the output of the scan chain [31]. Since internal nodes can be controlled and observed using this method, the test generation can be done using a combinational ATPG tool which is simpler than a sequential ATPG. Hence, a number of studies on improving scan chain structures have been proposed in the literature [33, 34, 35].

Although manufacturing test is a key step in accelerating yield ramp-up that grantees product quality, it cannot detect errors that have escaped pre-silicon verification and are of design nature rather than manufacturing detect. As stated earlier, manufacturing test, verifies the fabricated circuit against the implemented design and relies on the circuit netlist as the reference model. Therefore, design errors that have escaped pre-silicon verification also exist in the netlist and cannot be detected by manufacturing test. Post-silicon validation is a complementary step whose focus is on detecting those bugs that have still remained undetected.

1.5 Post-Silicon Validation

As stated in section 1.3, although pre-silicon verification offers full controllability and observability, it is known to be considerably slow when compared to the real-time execution; for example, register-transfer level (RTL) simulation speed is approximately 6 orders of magnitude slower than the execution speed in silicon prototypes [36]. Considering the rising trend in the size and complexity of modern designs, functional verification might require tens and even hundreds of person-years and the computing power of thousands of workstations [37]. Employing server farms in this step requires huge amount of energy and time for simulating different aspects of design, but even then, due to limited number of simulation cycles and state space explosion, certain aspect of the design cannot be verified in an acceptable practical time [36, 37]. In addition to this, in spite of all the verification efforts that are carried out before manufacturing, errors are still discovered in around two thirds of the silicon prototypes [38]. If SoCs that contain undiscovered subtle errors are delivered to the market and subsequently malfunction at some point during their operation, the outcomes will be unexpected and can impose significant operational and financial burden on the manufacturer, example of such is the Cougar Point SATA bug in Intel's Sandy Bridge chipset [39].

Due to lack of accurate delay timing models in pre-silicon verification, the correctness of timing characteristics cannot be ensured. Despite the availability of accurate tools that evaluate electrical features of the design such as crosstalk, certain electrical characteristics cannot be ensured due to the slow nature of simulation techniques to model these effects. When combining the above constraints with tight time-to-market windows, it is common practice that designs are taped-out whenever the verification confidence is deemed sufficient. Though before committing to high-volume manufacturing, some verification steps have to continue on the prototypes, a task commonly referred to as **post-silicon validation** (also called silicon debug[40][41]). Post-silicon validation is a complementary step after pre-silicon verification that can account for up to a third of the development time for a new design [38] and allocation of significant financial resources [42].

Although both post-silicon validation and manufacturing test deal with the manufactured chips, one should note the fundamental differences between the two. As it was articulated in section 1.4, manufacturing test is concerned with the detection of fabrication defects for every single fabricated chip. On the other hand, post-silicon validation aims to check the design correctness rather than the imperfections in the manufacturing process. Instead of validating all the manufactured chips, a few prototypes are selected and a large number of validation tests, ranging from random input sequences to end-user applications, such as operating systems, computer games or scientific applications, are applied to the silicon prototype and its behavior is monitored at runtime for unexpected events, such as system crashes or incorrect results [42].

There are two main types of design errors (or bugs) that escape to silicon: *functional* errors and *electrical* errors. Functional errors occur when the implementation deviates from the specification, e.g. the condition for a state transition was incorrectly described in the RTL code [43]. Electrical errors, on the other hand, are caused by subtle interactions between the design and the electrical state of the system, such as power supply noise or thermal effects [44]. Due to the approximations used in circuit-level modeling, as well as the computational requirements needed to account in sufficient detail for the device physics, it is difficult to discover all the electrical errors before tape-out. Therefore, they are the dominant type of design errors that escape to silicon. They emerge at apparently random times under unique and difficult to reproduce conditions. For instance, a sub-block of a circuit might exhibit faulty behavior only at a certain working temperature due the variations in power supply. This means that even if there exists a mechanism for controlling certain operating conditions such as the temperature, the electrical errors that occurred at a specific temperature might not be easily reproduced as there are many factors that contribute to the emergence of the error. These subtle electrically-induced errors commonly manifest in logic domain as bit-flips in flip-flops [44, 45].

It is important to stress that, unlike fault-tolerant mechanisms that are focused on fault containment and recovery [46], post-silicon validation aims at detection and localization of such errors. Subsequently, once the bug has been identified, steps can be taken in order to determine if it can be repaired[47], corrected by using software patches[48] or sent for a costly re-spin.

Limited observability makes error detection and localization cumbersome. Many design for debug (DfD) architectures and methodologies have been proposed to improve internal observability. For instance, embedded logic analysis [38] is used to record data on on-chip trace buffers. The collected data can then be transferred into the simulation frameworks for post-processing in order to locate the error. Another example is the method presented in [49, 50] to record instruction footprints for microprocessors. In all these methods, since the traces that are acquired on-chip are constrained in depth, it is critical to detect bit-flips within a small number of clock cycles after their occurrence in order to confirm that the recorded data on the trace buffers is relevant to the root cause of errors, both in terms of time and location of occurrence. For instance, as stated in [51], a chip with roughly 200,000 flip-flops running at 100 MHz can easily creates 20,000 Gigabits of information per second. Dumping this information in real-time for offline processing is impractical.

To reduce the *error detection latency*, which is defined as the amount of time it takes for an error to cause an observable failure, system-level methods have been investigated [5]. The core idea from [5] is to detect errors rapidly using time-redundant execution, which is diversified and combined with fine-grained checking. The transformation of validation tests is done in a systematic manner, nonetheless the method has been architected for, and hence restricted to, microprocessor-based designs. A more detailed explanation of the idea in [5] is discussed in chapter 2.

A fundamentally different, yet effective and promising direction of research for reducing detection latency, which is generic and not dependable on type and operation of circuit blocks, is to rely on embedded hardware monitors for run-time property checking. Assertions are properties that must be satisfied during circuit's operation[27] and are extensively used for detection and localization of bugs during the pre-silicon phase, known as assertion-based verification[52, 53, 54]. For instance, it has been reported that during pre-silicon verification of an industrial project, 85% of bugs were discovered using assertions [27]. In chapter 2, we will discuss assertion-based verification in more detail and provide an overview of how hardware assertions can be used in post-silicon validation. The latter provides the core information for the contributions of this thesis.

1.6 Contribution and Organization of the Thesis

Although using assertions in hardware enables the real-time checking for any circuit block, as it is the case during the pre-silicon phase, crafting these hardware assertions based on design functionality, rather than its structure, makes it difficult to assess how many of the potential bit-flips are monitored during a validation session. In this thesis, we propose several methods with the objective of identifying assertions that are most suitable for post-silicon validation.

Chapter 2, provides a more detailed discussion on background information and prior works in post-silicon validation with special focus on reviewing techniques for improving the error detection latency in post-silicon validation.

Chapter 3 introduces our proposed systematic methodology that automatically selects a subset of assertions for the purpose of detecting bit-flips in flip-flops during post-silicon validation. As part of our methodology, we have designed algorithms to rank assertions based on their potential to detect bit-flips. We have introduced two quantitative metrics: 1) *the bit-flip coverage estimate* and and 2) the *flip-flop coverage estimates* which can be used to assess the quality of the selected assertions. As at the time of writing this thesis, there is no well-defined and universally accepted coverage metric in post-silicon validation as opposed to pre-silcion verification (e.g., code-coverage) and manufacturing tests (e.g., stuck-at-fault coverage), introduction of these two metrics is a step forward towards establishing a unified metric. The contributions from this chapter appear in [55] and [56].

One of the shortcomings of the methodology that is introduced in chapter 3 is the fact that the evaluation of the potential of assertions is carried out in a simulation-based environment. Since post-silicon validation runs at silicon speed, closing the gap between the slow simulation speed and the fast silicon speed is of crucial importance. One common way to do this is to use Field Programmable Gate Arrays (FPGAs) as the intermediate platform [57, 58, 59]. Hence, in chapter 4, we will introduce a fully automated methodology that generates emulation-ready architectures that can automatically be integrated in the tool flow that is presented in chapter 3 to improve the accuracy of selection and assessment of hardware assertions. We will show that by incorporating emulation in our flow, the flip-flop coverage estimate (introduced in chapter 3) is improved on average from 38% (5 error injection) to 49% (5000 error injection). This improvement is largely due to using emulation that facilitates the discovery of more accurate relationships between assertions and flip-flops that can be covered by them. The contributions from this chapter appear in [60] and [61].

Finally, it is worth noting that, one of the main challenges that was faced during this work was the quality of the assertions in terms of detecting bit-flips in multiple flip-flops. As it will be elaborated in detail in chapters 3 and 4, GoldMine [62, 3, 63] was used as the assertion generation tool in the first two contributions of this work. The machine learning approach in GoldMine which is based on the simulation traces is not intended on generating assertions that specifically target post-silicon validation. Since post-silicon validation is done after the design is taped-out and the netlist is ready, the automatic assertion generation tool should ideally be based on the structure of the netlist without relying on the simulation traces. In addition, we realized that the internal mechanism of Boolean Satisfiability Solvers (SAT) which will be discussed in detail later on, can be leveraged to extract relevant information from which hardware assertions can be implied. Hence in chapter 5, we present a fully automated SAT-based methodology for fast generation of assertions by using the built-in pruning mechanisms within SAT solvers, namely learned clauses. These assertions are evaluated for their potential to detect bit-flips using a new incremental SAT-based approach. In addition to speeding-up the simulation-based approaches for assertion generation and evaluation, when compared to the known art, our results show improvements in both the number of flip-flops that can be covered for bit-flip detection, as well as for the on-chip area for the bit-flip detection unit.

Finally in chapter 6, we summarize the technical contributions of this thesis and provide possible directions for future research.

Chapter 2

Background and Related Work

Unlike pre-silicon verification and manufacturing test that have benefited from many improvements in the tool flow and algorithmic methods which has gradually resulted in development of mature automated methodologies, post-silicon validation is still relying on *ad-hoc* methods with a few automated methodologies that are available only for specific application domains such as validation of microprocessor designs [5, 49, 50]. However in modern designs, there are many complex digital blocks which are not fully controllable/observable by microprocessors, yet there are no systematic methods to validate them. Hence, the contributions from this thesis aim to address this issue.

As seen in Figure 2.1, post-silicon validation can be divided into sub-areas. In this chapter we provide relevant background information regarding stimuli-generation which is focused on the controllability, root-cause analysis which belongs to the error localization, and finally error detection, which tackles the limited observability and is the main focus of the contributions from this thesis.

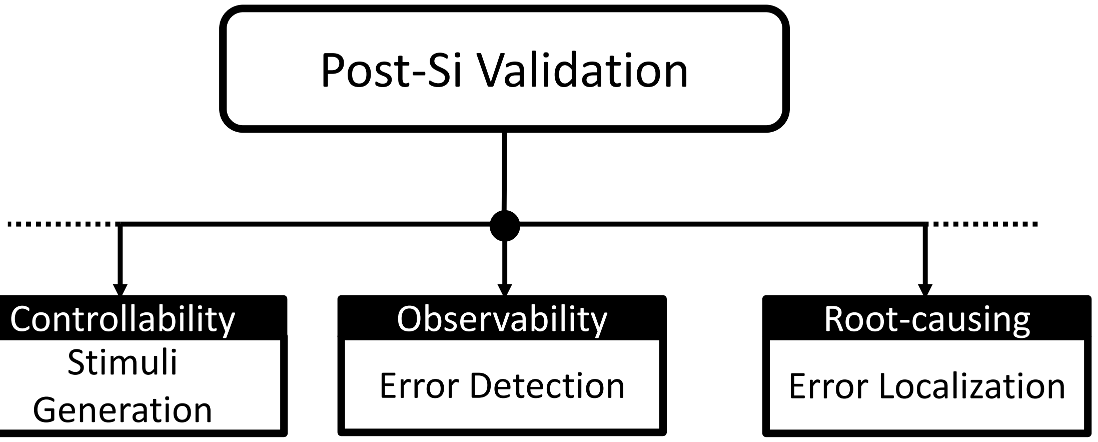


Figure 2.1: Different sub-areas in post-silicon validation.

2.1 Controllability

The term *controllability* which is frequently used in digital verification is conceptually derived from the same term used in control theory [64]. It is essentially concerned with applying a set of controlled input vectors that push the circuit into a desired state. Therefore, before discussing the methods used for stimuli generation in post-silicon validation, a quick review of the same practice in pre-silicon validation would be provided.

2.1.1 Stimuli generation in pre-silicon verification

As discussed in section 1.3, functional simulators are used during pre-silicon verification stage. In order to verify the functionality of the design-under-test (DUT), a large set of input vectors are applied to the design and the design's response is checked against the expected results based on the specification from which the implementation is based on. Depending on the functionality of the DUT, the stimuli can be an executable program for microprocessors, protocol-specific data packets or digital sample models for digital signal processing blocks. The stimuli can be manually prepared to check a specific feature of the design by guiding the circuit to a very specific state and check certain features, a method that is commonly referred as directed tests[65, 66] or could be automatically generated based on some constraints to produce valid and functionally compliant inputs to help uncover unforeseen problems (constrained random tests) [65, 67, 68]. The randomization is based on the constraints written in hardware description languages that are supported by most commercial simulators. The quality of the stimuli generated from either of these two methods is assessed using coverage metrics such as code coverage (statement coverage, branch coverage, etc.,) [24, 69, 70, 71, 72] which can provide useful feedback to the quality of the applied input vectors.

2.1.2 Stimuli generation in post-silicon validation

Similar to the pre-silicon verification, using directed tests to validate the functionality of the circuit under validation (CUV) requires the test cases to be tightly coupled with the design functionality which results in poor re-usability from one design iteration to another. To address this inflexibility, one can use randomly generated inputs that are constrained to be *functionally compliant.* During post-silicon validation, a large number of these constrained random sequences are applied to the design in order to detect design errors that have escaped pre-silicon verification and appear in silicon prototypes [73, 44, 74, 75]. Since due to bandwidth limitations, it is impractical to transfer the generated random input sequences in software to the hardware prototype, dedicated architectures must be developed for on-chip generation of constrained-random input stimuli for post-silicon validation. Several methods have been provided for microprocessor-based designs such as using instruction-level templates to guide the sequence generation [76, 74] or generating executable machine code that stresses the block under validation [50, 49]. In addition some recent research works have focused on addressing the issue of random generation of constrained inputs for generic digital blocks (e.g., video codecs, power management units etc.) [77, 78, 79]. The specific details of the above-mentioned methods are beyond the scope of the work from this thesis, which is focused on improving *error detection*. The interested reader can refer to the several references that were provided in this section.

2.2 Observability

Similar to the controllability discussed in the previous section, *observability* is also derived from concepts in control theory [64]. It is concerned with ensuring that the *response* of the circuit is consistent with the expected results that are based on the specification. Since some of the methods that are used in pre-silicon verification can also be adopted for post-silicon validation, some known methods that are used in pre-silicon for improving observability are discussed in section 2.2.1. Afterwards, a discussion on the prior works in this area for post-silicon validation is provided, which helps position the contributions from this thesis.

2.2.1 Observability in pre-silicon verification

Traditionally, the response from a reference model, known as *Golden response* was used to compare the observed output while applying the input sequences. However, even in cases where the observed response (typically at primary output) is identical to the golden response, it is not guaranteed that the internal signals of the design would hold the correct value [80]. This is because some errors could be masked before reaching the primary outputs. While it is possible to refine the reference model to account for internal signals, this would make the process slower to the point where it becomes practically infeasible to compute the golden response [80].

Another technique that has been commonly adopted over the past decade is to use assertions during pre-silicon verification which is known as assertion-based verification (ABV) [81, 82, 27]. As it will be discussed in detail in section 2.3, assertions are properties that need to be hold true indefinitely [27]. These properties are derived based on the specification and are independent from the specific implementations. This means that an exhaustive set of assertions that cover different requirements of the specification can be used when a large number of constrained-random input sequences are applied to the circuit during which the assertions should not violate. Should an assertion be violated, it represents an underlying problem in the implementation or an ambiguity in the specification which needs to be corrected. Information from a violated assertion further improves the internal observability in pre-silicon verification.

2.2.2 Observability in post-silicon validation

Unlike pre-silicon verification that offers the inherent ability to probe all the internal signals using functional simulators, observability is known to be one of the difficult challenges in post-silicon validation due to the limited access to the internal signals of the manufactured hardware prototypes. Physical probing of the internal signals has been introduced as a method to address this issue by probing the internal signals and post-processing the acquired data to find out the root-cause of the problem [83, 84, 85, 86]. However, due to the complexity of the modern ICs, a localization step needs to preceded the failure analysis. This localization step will determine the subset of the internal nodes that need to be physically probed to enable root-case analysis by correlating the simulation data with the observed data on silicon (I/O ports, internal nets) [87]. This is achieved by adding Design-for-Debug (DFD) structures to capture the internal states of the circuit under validation (CUV). In section 2.4, a detailed discussion of the some of the well-known DFD structures that are used to facilitate internal state dumps will be provided. Nevertheless, all these methods start to dump the internal data when a malfunction has been observed e.g., a system crash. Therefore, these DFD structures become unsuitable for providing relevant information for pin-pointing the subset of the design where the error has initiated from if the errors manifest after a long period of operation [88]. Hence, the latency from the time the error has been sensitized to the time the state is dumped, namely *error detection latency* must be minimized. Facilitating automated methodologies to design DFD blocks that detect errors as soon as they occur is the main motivation of the contributions of this thesis. Hence, in section 2.5, some of the recent efforts that are available in public domain which address the problem of reducing *error detection latency* are discussed and the preliminary background regarding the idea of using hardware assertions for reducing error detection latency in post-silicon validation is provided.

2.2.3 Root-cause analysis

Across both the pre-silicon verification and post-silicon validation stages, once an erroneous behavior is identified, the quest for finding its root-case begins [89]. Following to the detection of an error, the sequence of stimuli that has been applied to the CUV as well as the traces that have been collected from the internal nodes are analyzed to diagnose the errors. This can involve the iterative process of reproducing the suspicious events and analyzing the CUV's response until the error is isolated. At this stage, some pre-silicon verification techniques can be reused to aid the diagnosis. For instance, pre-silicon simulation tools can be fed with captured traces to replay the erroneous behavior [90]. In addition, formal methods can be used to identify the root-case of the error [91]. As stated earlier, root-cause analysis is an important step that is the key to identifying the culprit of errors that have been detected. However, the detailed discussion of the state-of-the-art methods and techniques in this area are out of the scope of this work and the interested reader can refer to [89, 90, 91, 92, 93].

2.3 Assertion-based verification

As it was mentioned in section 2.2.1, assertion-based verification (ABV) is used as one of the key methods to improve observability during pre-silicon verification stage.

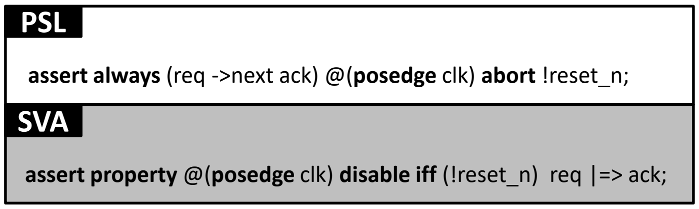
An assertion (also known as an invariant) is a behavioral statement that can be used in any of the following scenarios [94]:

- A specification that outlines the required behavior.
- A statement that determines the assumptions about the system inputs.
- A formal statement that can be checked using formal verification tools.

- An active element that can check for design errors during simulation.
- A series of statements that determine the interface specifications for Intellectual Property (IP) reuse.
- An item for determining coverage in pre-silicon verification.
- An item in the verification plan.

Therefore, an assertion is a description of the design property which needs to hold true indefinitely. A good comprehensive set of assertions can be considered as comments on the partial expected behavior of the design which can serve as a potential documentation of the target designs [95].

There are two commonly used languages for describing assertions: Property Specification Language (PSL) and System Verilog Assertions (SVA). Figure 2.2 shows an example of a SVA and a PSL property that assert the the following property: when the active low reset signal (`reset_n`) is 1 and the request signal (`req`) is also 1, the acknowledge signal (`ack`) should become 1 in the next rising clock edge.


PSL

```
assert always (req ->next ack) @(posedge clk) abort !reset_n;
```

SVA

```
assert property @(posedge clk) disable iff (!reset_n)  req |=> ack;
```

Figure 2.2: An example of a PSL and a SVA assertion describing an identical behavior.

It is important to note that an assertion does not contribute to the circuit that is being designed but rather is a mechanism to ensure that the final design meets its intended behavior. When an assertion fires, it provides useful information about when and where the property is violated and what has been the sequence of events that pushed the circuit to the point where a property was violated. Since the assertions normally contain internal signals as well as primary inputs and outputs, the observability in case of an error is improved. ABV's effectiveness in detecting errors, specially those that are triggered in subtle corner-case conditions have lead to its adoption in practice. For instance, statistical results from an industrial project shows that up to 85% of bugs were detected using assertions [27].

Traditionally assertions have been developed by design and/or verification engineers which clearly imposes significant burden to write comprehensive assertions that cover all different angles of the functional behavior of the design. This effort becomes a daunting task when there is a change in the specification or implementation that requires all the pre-developed assertions to be revisited. Therefore, similar to the different phases of the design flow, having automated methodologies for generating assertions would be very beneficial. There has been a number of studies for automatic generation of RTL assertions that are pervasive and independent from designer's knowledge of the design. Authors in [2] introduce an automatic tool for Implementation of Dynamic Invariant Extraction, hence the name IODINE, to infer likely invariants based on the design's simulation traces. The IODINE framework is shown in Figure 2.3. The design is initially simulated using extensive random or direct tests and the simulation data is extracted in a format such as Value Change Dump (VCD) that is independent from the HDL language used for implementation. It is assumed that the simulation traces are extracted from a test which is known to pass to ensure that the invariants are inferred over the correct functionality of the design. The analyzer considers the simulation dumps and uses a process that

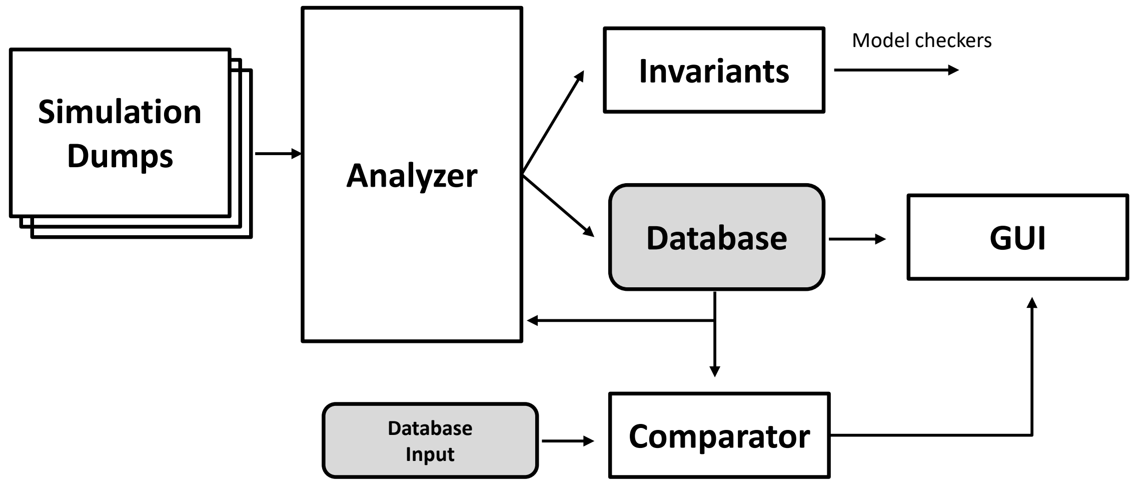


Figure 2.3: IODINE Framework [2].

determines some common signal relations such as OneHot analyzer for finding a set of signals that are one-hot (only one signal is 1) or one-cold (only one signal is 0), Mutex analyzer which finds signals that are mutually exclusive and State analyzers which record state transitions based on visited states and conditions for transitions. The set of approved invariants are added to the database input and the newly found invariants can be compared against the older ones using the provided GUI interface. Although this method provides automatic extraction of properties, it's limitation in terms of covered behaviors in the analyzer as well as the dependency on tests that need to be checked for correctness make it un-scalable for designs that are constantly changing.

In a different study, authors in [96] provide a methodology that uses the already verified invariants to produce more complex set of invariants. Similar to IODINE, the properties are generated based on a snapshot of simulation traces. The process of generating the properties has multiple phases. At the first phase, a set of basic properties are searched and added to the database. In the following phases, these properties are used for developing more complex properties that cover more functional space of the design. The process continues until no further property is found. Yet the limitation of this methodology is also on the limited operators that are supported. Based on [96], the supported checkers are from Open Verification Library (OVL) standards and are limited to OVL_Increment, OVL_OneHot, OVL_Handshake and Req-N-Grant.

The work in [95] introduces an interesting approach in using Quantitative Boolean Formula (QBF) solvers [97] based on analyzing formulations from Look Up Tables (LUTs) inserted in circuits that are solved using incremental Boolean satisfiability solvers [98]. Signals from different parts of the design are selected and passed to LUTs of different size to see if a logical relationship can be inferred in between these signals. As mentioned in [95], this synthesis problem can be formulated as QBF which is essentially the same problem as logical debugging introduced in [99]. As authors have also mentioned, although this work provides a promising direction for using Boolean satisfiability solvers for automatic generation of hardware assertions, the set of inferred assertions are found between a set of pre-given signals. In order to avoid random selection of these signals, heuristic algorithms needs to be developed to carefully choose a subset of signals for which the inferred assertions would target a specific goal.

A more recent work on automatic hardware assertion generation is GoldMine [3, 62] and its framework is shown in Figure 2.4. GoldMine uses data mining and machine learning algorithms to produce likely invariants that can be verified using a formal verifier that dismiss the wrong invariants. As shown in Figure 2.4, the design (RTL or gate-level format) is fed to the static analyzer and the data generator blocks.

The static analyzer extracts information from design that can aid the mining engine to reduce the chance of false positives. The static analyzer tool used in GoldMine is a cone of influence technique to select a set of feature variables for a given target variable. Since data mining relies on statistical methods to infer relationships, one variable can be associated with another variable even though the second variable cannot affect its value. By using the static analyzer, this problem is eliminated.

The data generator box simulates the design using random inputs with some configurations such as the number of clock cycles for which the design is simulated. In case the simulation traces has been pre-computed based on a specific functional scenario, the data generator step can be bypassed and the simulation traces in the VCD format is provided to the mining engine. The mining engine searches the simulation traces to infer relationships between the feature variables and target variables. It has different mining algorithms such as Decision Tree, Decision Forest, Prism and Coverage Miner. Details about the specific details of the algorithms are out of the scope of this thesis and one can refer to [3] for more information. Finally, the formal verifier will use a commercial formal verifier (Cadence Incisive formal verifier [100]) tool to ensure the correctness of the likely invariants. In addition, for false invariants, the counter examples can be fed back to the miner to refine the simulation traces. The

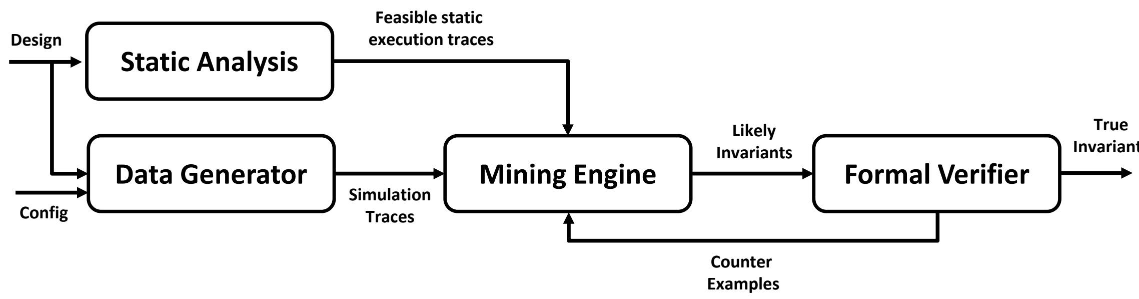


Figure 2.4: GoldMine Framework based on [3].

invariants that are confirmed by the formal verifier can then be used for pre-silicon verification purposes or as it will be shown in the first two contributions of this thesis, for post-silicon validation purposes. While GoldMine provides useful invariants that can be used for regression and provide feedback on the quality of the input stimuli based on analyzing the number of times each assertion is exercised, its dependency on simulation traces makes it susceptible to finding trivial assertions in case the simulation traces do not cover a wide range of functional aspects of the design. As it will be discussed later in chapter 5, this is a significant drawback for post-silicon validation and is one of the motivating factors for our proposed methodology for automatic generation of hardware invariants that are tuned for post-silicon validation. In addition, using a two-step methodology which relies on using a formal verifier at the second phase further aggravates the limitations of the tool.

In this section, we reviewed some of the state-of-the-art automatic assertion generation tools and discussed their limitations. In the following sections, we will show some of the common methods to improve observability in post-silicon validation within which the idea of porting assertions to post-silicon validation to improve the internal observability is introduced.

2.4 DFD structures to improve observability

As it was discussed in the earlier sections, unlike pre-silicon verification which benefits from significant flexibility in observing internal signals via probing internal signals or by using assertions, internal observability is one of the most difficult challenges in post-silicon validation. Hence in this section, we review the existing methods to address this problem.

2.4.1 Scan chain-based technique

The main idea in scan chain-based techniques is to reuse the internal scan chains that are available as the most widely used structures to increase observability during manufacturing test [101, 102]. Though, the scan structure must be modified to allow the scan infrastructure to support three basic features: scan for access, breakpoints and clock control. As explained in section 1.4, scan flip-flops (SFFs) are connected together to form a shift-register structure called a scan chains. During test mode, the data vectors are shifted into the scan chains through the scan input pin and the circuit response is shifted out through the scan output pin for pass/fail analysis as shown in Figure 2.5. On the other hand, during post-silicon validation, functional data must be supplied to the circuit under validation (CUV). Therefore, the access mechanism of the scan infrastructure must be modified as shown in Figure 2.5. As it can be seen, the scan chains are connected together to form a single long scan chain and the access to the chain is done through low bandwidth dedicated pins such as the test data input (TDI) port of the JTAG interface [103]. It is important to note that this approach is commonly used when low cost debug equipments are employed. In case of using more advanced debug equipments with the ability to control multiple scan-chains, the choice to tie all scan chains together can be avoided.

Due to the fact that it will take multiples of clock cycles to offload the data from SFFs, by using the scan chain structure, the data should only be inserted into the scan chain when a particular event of interest has occurred. The consecutive data acquisition can start after the current data in scan chain has been completely offloaded. One additional hardware structure that can be inserted in the CUD to indicate a single event from which the acquisition should start is called the breakpoint control

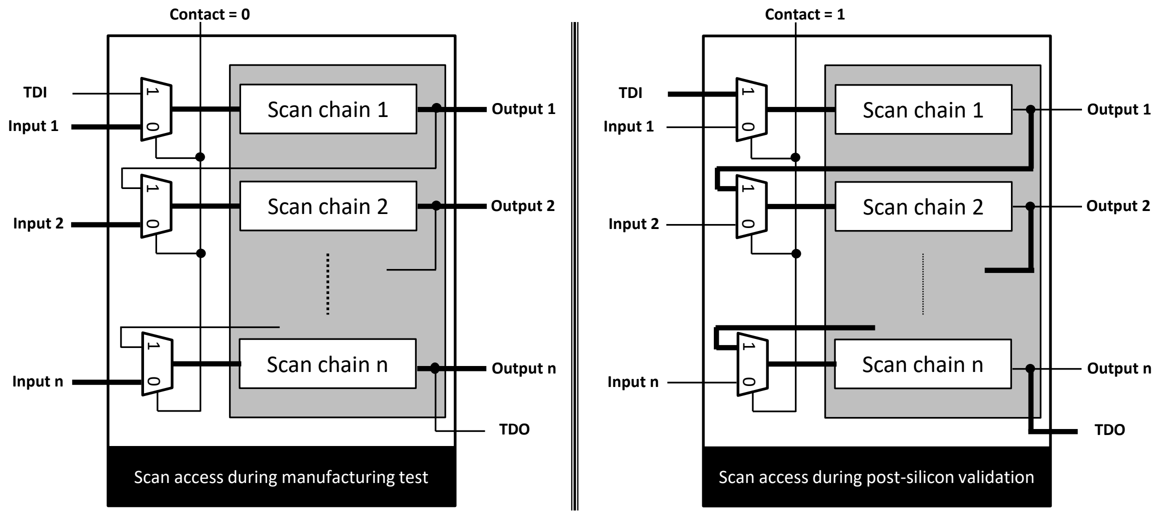


Figure 2.5: Differences in scan architecture in manufacturing test and post-silicon validation [4].

unit. It is usually built with comparators that monitor a set of trigger signals e.g., program counters or internal instruction/data buses [104]. When the value of the signals coming from the CUD matches with the breakpoints conditions, the breakpoint flag is raised. During post-silicon validation, the breakpoint controller can be configured using a serial interface such as JTAG [103] such that different scenarios are covered.

When the breakpoint flag is raised, the circuit operation must be stopped to preserve the current state of operation. This can be facilitated by creating a gated clock unit which provides different clock signals depending on the mode of the operation as shown in Figure 2.6. In the halt mode, the breakpoint condition has been reached and the on-chip clock is stopped. In the scan mode, the debug clock is passed to the gated clock output for loading the scan chains shown in Figure 2.6. Likewise, in single step mode, a single clock pulse is generated at a time to step through the normal execution of the chip. Clearly, during the functional mode, the output of the

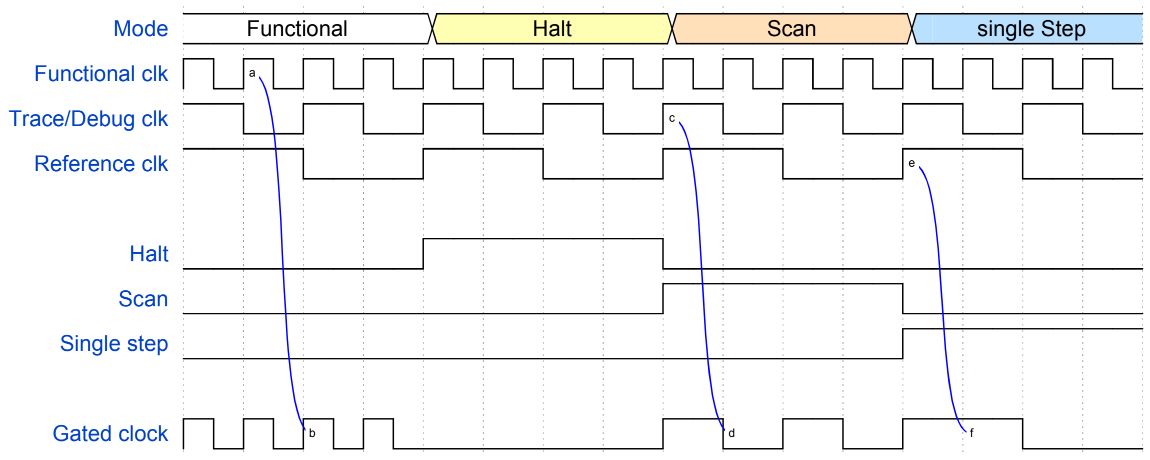


Figure 2.6: Clock control timing diagram for different operation modes.

gated clock must be the same as the on-chip clock.

Despite of some proposals for improving scan chain-based methods for improving observability, the main drawback of using this technique is the fact that during post-silicon validation, the circuit has to be stopped when offloading the data from the scan chains which means that it is not feasible to acquire data in real-time. As stated before, some logic bugs manifest after thousands of clock cycles [105] which makes those methods which do not halt the circuit operation more favorable. Although this can be done by double buffering the scan elements, the resulting area overhead might be unacceptable in practice [106]. Even in the case where this penalty is acceptable, data sampling in two consecutive clock cycles would not be possible because multiple clock cycles are needed to scan out the acquired data in the shadowed buffers. The ability for continuous data acquisition is an important requirement for isolating errors that manifest over a large number of clock cycles.

2.4.2 Trace-based technique

The motivation for using trace buffers stems from the fact that the circuit execution does not have to be interrupted which enables real-time data acquisition [107]. The idea has been successfully used in microprocessors and embedded systems to enable software debug [108, 109, 110]. An interesting point to mention is that the bug hunting during post-silicon validation which is essentially hardware debug, follows a similar approach in that the debug process involves analyzing data that has been acquired after a specific event has occurred. Therefore, logic analysis has emerged as a practical method for the debug of microprocessors [111, 112], FPGAs [113, 114, 115] and complex SoCs [38, 116, 117].

By using logic analyzers, the acquired data can be extracted through the circuit pins by using external logic analyzers. However, with the constant increase in design complexity, logic-to-pin ratios would also rise which suggests the functionality provided by the external logic analyzers should be embedded into the design.

Figure 2.7, represents an example of en embedded logic analyzer (ELA) architecture. The ELA can be categorized into 4 different units: control unit, sample unit, offload unit and the trigger unit. The latter is of particular importance to the work in this thesis as the contributions provided are placed into this trigger unit. The control unit is composed of a finite-state machine (FSM) with a programmable register bank that can be modified through a serial interface. The control unit is responsible for facilitating different trace acquisition sessions, choosing the events and initiating the data offload. The sample unit is controlled by the control unit and stores traces from a set of trace signals into embedded memories that are clocked similar to the host clock. The serializer unit reformats the data in such way that they can be transferred via low-bandwidth device pins. Finally, the trigger unit is responsible for monitoring the trigger signals through one or multiple event-detectors and communicate with the control unit when an event of interest occurs after which the control unit would organize the offload sequence of traced data through the serializer in offload unit. The design of the ELA unit is done at the chip design phase at which time the decisions about the type of the event-detectors, signals that are connected to the trigger unit and the sample unit etc., are made. Once the chip is brought up during validation, the ELA will work at the same time as the host during which it offers real-time monitoring of the behavior. Should an abnormality happen, the traced data in embedded memory buffers will be offloaded and analyzed at the root-case analysis phase. The detailed discussion of the different components of the ELA unit is out of the scope of this work and the interested reader is referred to [107, 4, 118]. Since the contributions of this thesis are placed within the scope of the trigger unit which enables

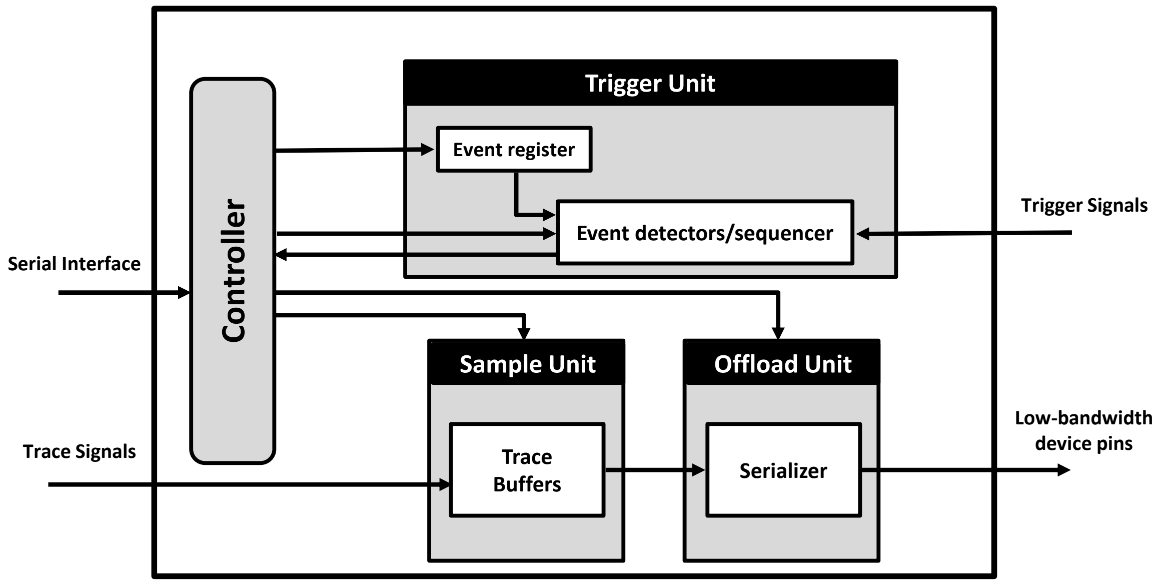


Figure 2.7: Example of an embedded logic analyzer unit introduced in [4].

event-detection, a brief review of some of the known works in this area with their pros and cons are explained in the next section.

2.5 Event detection

Event-detection as the core of the trigger units is an important part of the ELAs. An efficient event-detector will detect errors as soon as they occur so that the content of the trace-buffers are most relevant to the sequence of events that led to the error, hence increase the chance of error localization at the root-cause analysis steps. There are two fundamentally different approaches to accomplish error detection; a) by modifying the validation tests and b) by adding on-chip hardware circuitry inside the ELA. The former is detailed in section 2.5.1 and the latter in sections 2.5.2 and 2.5.3.

2.5.1 Quick error detection tests

A fairly recent work which offers error detection with little to none modification in hardware is to modify the validation test cases in such way that the failures are detected through the built-in fine-graded checking mechanism of the modified tests [5, 119, 120, 121, 122]. The idea has been inspired from the concurrent error detection in fault-tolerant computing [123, 124, 125] bearing in mind the distinct differences between fault-tolerant computing and post-silicon validation which are 1) Unlike fault-tolerant computing, there is no need for recovery from a fault in post-silicon validation and 2) Performance penalties in the sense of execution time are acceptable so long as it reduces the debug time which is the most time-consuming task in post-silicon validation.

The existing validation tests can be modified to Quick Error Detection (QED) tests in two ways as fully explained in [5, 121]. The first approach is to use *Duplicated instructions for validation* as shown in Figure 2.8. As seen, this method works

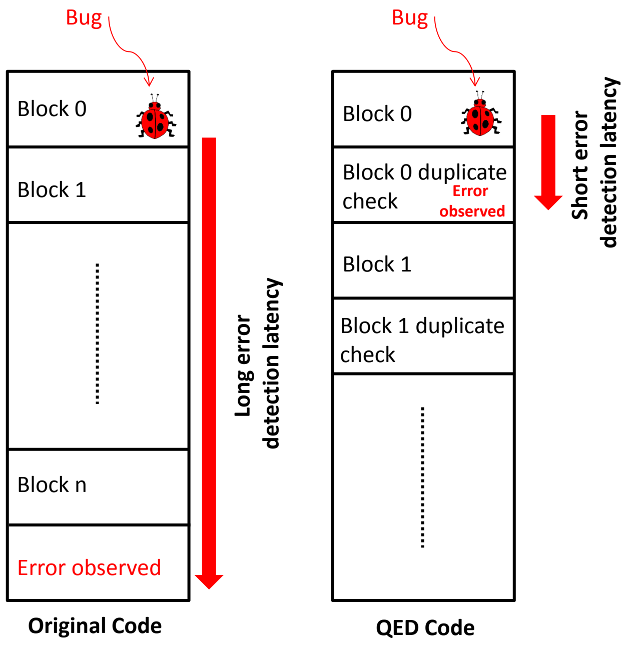


Figure 2.8: Using duplicated instructions to shorten error detection latency which is introduced in [5]

by duplicating each block of instructions and comparing their results immediately. Instead of waiting until the error reaches an observable point (e.g., program crash), the error detection latency in this case is defined as the sum of the time elapsed between the start and end of a single block of instructions and the time it takes to perform the check. The intrusiveness of the test, which is defined as the amount of deviation in the execution behavior of a QED test from the original test can be configured based on some parameters that can be used to modify when and where a QED test is inserted. By using this method, half of the memory space and general purpose registers are reserved for the original instructions and the other half for the

QED tests. Some special considerations for specific scenarios such as loops, conditional statements etc., must be placed and the reader can refer to [5, 121, 120] for more specific details.

Another approach for modifying validation tests to QED tests, as shown in Figure 2.9, is to use the *Redundant multi-threading for validation* transformation which is also inspired from fault tolerant computing [126, 127]. Instead of executing the original and duplicated instructions on the same thread, this approach executes the original instructions on one thread and uses another thread for the duplicated instructions as well as the check mechanism. Since the original block of instructions can be independently executed on a separate core, using this approach will offer reduced intrusiveness compared to the previous one. The error detection latency in this case is bounded by the time for the main thread to execute a block of instructions plus the delay it takes for the main thread to store the results and for the for the check thread to perform the comparison. By introducing a FIFO to store the intermediate results of the main thread, this approach can be modified to be a software-hardware modification.

Nevertheless all QED-based approaches target designs that execute on machine code and provide unclear support for peripheral units. Therefore, by using the approaches described in this section, one can significantly reduce the error detection latency in microprocessor-based design where instruction code is heavily used. However, since today's design contain many blocks that are generic digital blocks (video decoder, power-management IC, accelerators, etc.,), the need for formulating a different strategy for the validation of these blocks is justified. As it has been mentioned in several places in this thesis, the contributions of this work target designing automated

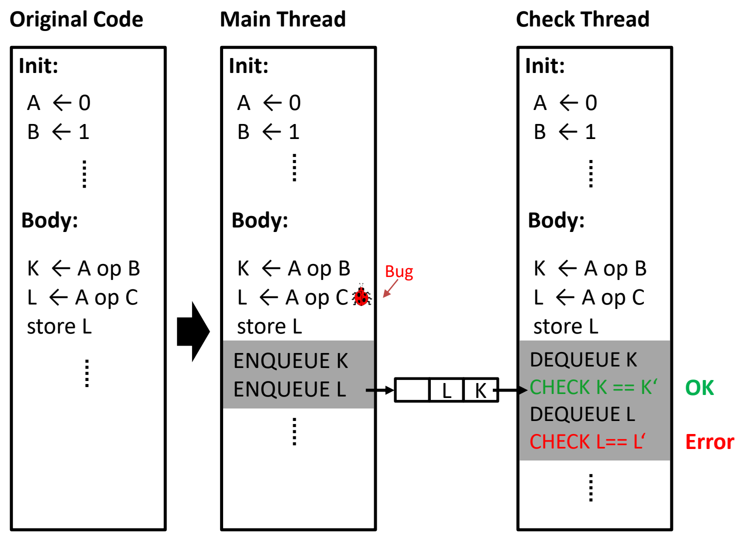


Figure 2.9: Redundant Multi Threading QED Transformation introduced in [5]

methodologies that aim at reducing error detection latency for these generic blocks. We will consider prior works in this area in the next two sections before discussing the main contributions of this thesis.

2.5.2 Programmable trigger-units

One of the directions in using hardware-based trigger-unit architectures is to have programmable trigger units. Though, the platform on which the design is implemented affects the way the trigger unit is designed. In FPGAs, the FSM and the glue logic around it can be modified with respect to the trigger conditions [114] which enables any trigger condition to be programmed. This gives the user the highest flexibility in terms of programing different trigger conditions during post-silicon validation. Therefore, so long as the recompilation time is reasonable, highest extent of flexibility can be achieved. Quite on the contrary, since in application-specific integrated circuits (ASICs), the design is manufactured in silicon, one will not have the ability to modify the design with a different trigger condition for a different debug session. Instead, the post-silicon engineer has to re-program the register banks in the control unit as well as the event-registers shown in Figure 2.7 for a different debug session. Nevertheless, the extent of events that can be programmed are limited. Therefore, to address this limitation, the idea of programmable trigger units (PTEs) have been introduced in [38]. A small portion of the logic with limited ability to program is placed inside the ASIC in such way that the inputs to the PTE is a multiplexer network that is connected to different set of trigger signals. The limited programmability, as explained in [38] comes from the fact that only one FSM of certain type can be programmed using the PTEs. Nevertheless, through a group of counters, timers and comparators, more complex trigger conditions can be programmed to the ASIC.

Automated methodologies for design of trigger units have been introduced in order to reduce the manual effort during post-silicon validation [38, 128]. For instance in [128], the trigger conditions are formulated in concise descriptions. The automated solutions can then analyze the triggers and map them to the available counters, comparators etc., in the PTE unit.

2.5.3 Assertion-based trigger units

As explained in section 2.3, there is a rising trend in popularity of assertion-based verification (ABV) in pre-silicon verification [27, 129, 130, 131]. During design verification, an assertion will fire if its corresponding property is violated. Yet the same idea can be brought-in to post-silicon validation to ensure those properties hold during at-speed circuit execution when a significantly larger volume of input stimuli is applied. But, pre-silicon assertions in their original PSL or SVA format are not synthesizable. Therefore, in recent years, a number of studies and projects have addressed the problem of generating the hardware equivalent circuit of the pre-silicon assertions [132, 133, 54, 134, 135]. This motivates the idea of embedding the hardware unit of these assertions on-chip as on-chip assertion checkers.

For instance, in [136, 137] the synthesized assertion module is coupled with a so called-assertion processor. They focus on how the circuit interacts with the assertion processor but provide no detail on how the processor is embedded in the overall chip architecture and how the external access to the assertion processor is enabled [7]. In [138], a mechanism is explained on routing internal signals to the outside of the chip to be monitored by an FPGA-based assertion checker, though this method will be inefficient for state-of-the-art high-speed processors and SoCs. In a different study[6], assertions were embedded on-chip in a reconfigurable block with a trivial area overhead in order to facilitate the organization of different post-silicon debug sessions in a time multiplexed manner as shown in Figure 2.10. Their results in a case study on hardware implementation of a H.264 decoder show that by using their proposed method, the error detection latency is reduced by 87 times. Needless to say, this approach is only applicable when a reconfigurable block is available on-chip.

Authors in [7] introduce several architectures for integration of hardware assertion checkers in SoCs that provide post-silicon debug support for both the scan-chain based techniques and trace-based techniques. Their proposed architecture supports enabling and disabling assertion checker units because when a SoC is shipped with embedded assertions, it is not desirable for all the assertions to be enabled. The common user of the SoC should might not be interested to be warned about different assertion

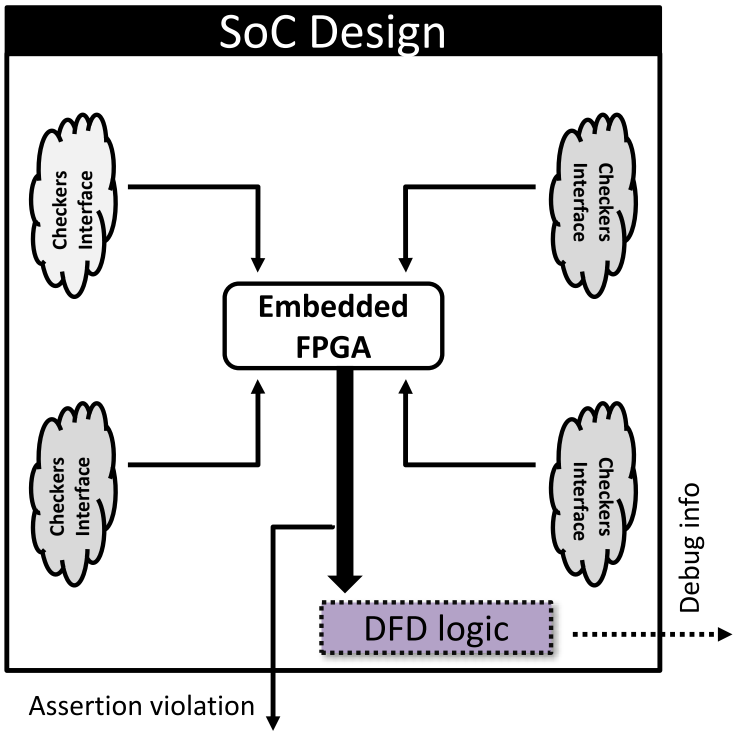


Figure 2.10: Using hardware assertion checkers with a centralized embedded FPGA introduced in [6].

violations. In addition, in power-sensitive designs, the power dissipation associated with the assertion unit is of great importance. However, when the SoC is sent back to the manufacturer because of a defect, the debug engineers will try to reproduce the problem by enabling the assertions. This control for hardware assertion checkers using debug registers is shown in Figure 2.11. The debug registers form a chain that is accessible via the Test Access Port (TAP) controller. Clearly, it would be more efficient to group related hardware assertions together and control them with a single debug register.

As mentioned before, the synthesized digital block of the assertions can be used inside the trigger unit of an ELA to facilitate error detection. When an assertion is

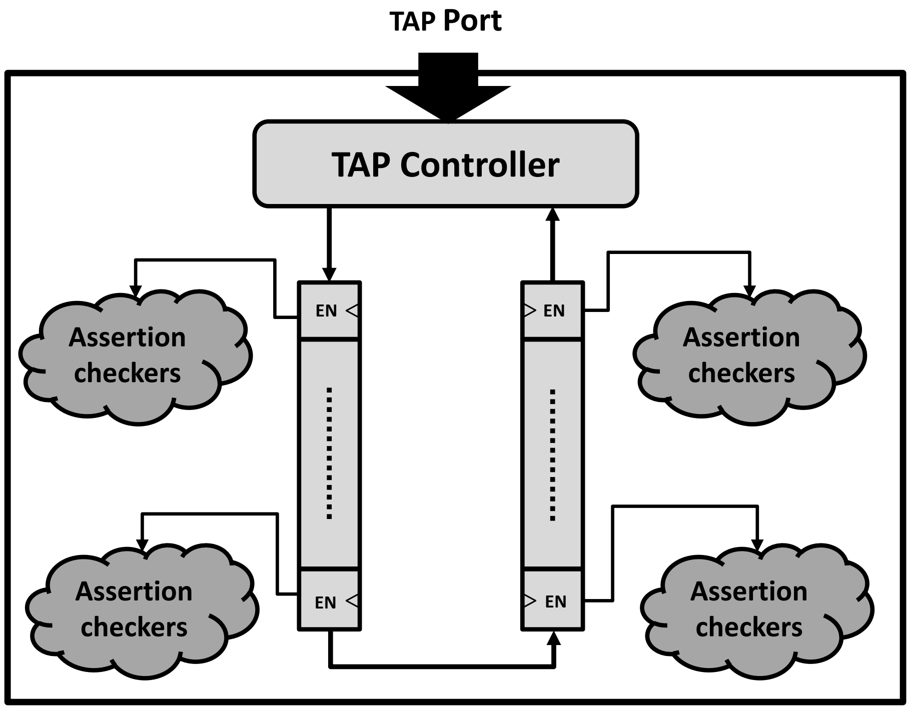


Figure 2.11: Debug control registers for enabling and disabling hardware assertion checkers [7].

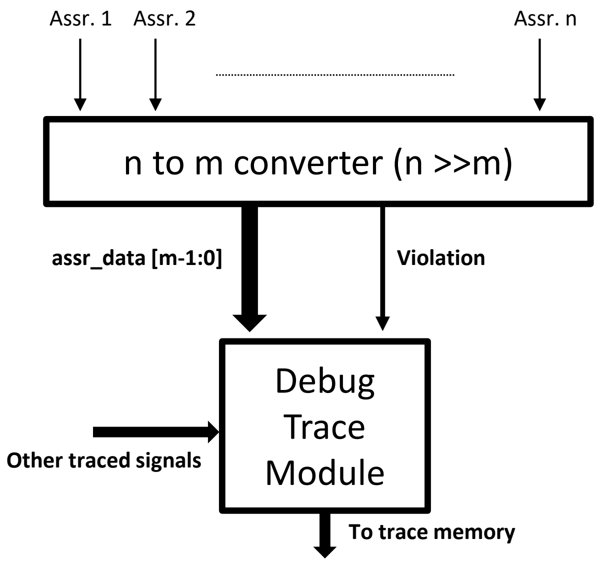


Figure 2.12: Capturing violated assertions using debug trace [7].

violated, the recorded data on trace-buffers can be offloaded for root-cause analysis. For instance, authors in [139] introduce a tool which generates hierarchical triggers which also provides compact trace information for root-case analysis. However, as stated in [7], one problem in most of the ELA based debug structures is the limited amount of available trace memory as well as the width of the trace channel in comparison with the total number of assertions in design (hundred or even thousands). Hence, assuming that there are n available assertions, it must be converted to a `assr_data[m-1:0]` signal that is compliant with the width of the embedded memory. The $n - to - m$ converter also provides a 1-bit `violation` signal that informs the debug module that there is a violation in the assertions to start trace acquisition.

In all the above mentioned works, the assertions are normally generated manually. This means that there is no automated methodology that systematically generates, evaluates and selects hardware assertions checkers that can be embedded in the ELA to enable error-detection. Therefore, through contributions of this thesis, we aim to address this limitation.

2.6 Summary

In this chapter, an overview of the existing post-silicon validation techniques with the main emphasis on real-time observability was provided. It was discussed that embedded logic analyzers (ELAs) are one of the most widely studied DFD techniques to improve observability and provide support for root-case analysis. As it was explained, improving the trigger-unit in the ELA is of high importance. This is because of the following two factors:

1. An efficient trigger unit must be able to detect as many errors as possible.

2. To aid root-cause analysis, the error detection latency must be kept short in order to record relevant data to the error occured in the embedded trace memories.

It was shown that assertion-based debug has become a popular method for design of DFD blocks. In particular, hardware assertions can be used as on-chip monitors in the event-detection unit of the ELAs. Though, little information is available in the public domain regarding the automatic extraction of these assertions from the netlist. Since post-silicon validation is performed after the netlist of the design is ready, designing automated methodologies that can identify hardware assertions based on the structure of the circuit would be very beneficial. Hence, in the remaining chapters of this thesis, the three contributions will focus on crafting automated methodologies for designing embedded architectures that rely on hardware assertions, that are automatically generated based on the design netlist, for detecting errors in post-silicon validation. In section 3.1.1, we will discuss in detail the motivation behind using hardware assertions for detecting errors that commonly occur in post-silicon validation.

Chapter 3

Automated Design of Embedded Bit-flip Detectors

As mentioned in chapter 2, to increase the relevance of the recorded data in the trace memories of embedded logic analyzers with the errors, the error detection latency must be minimized. In this chapter of the thesis, we will first introduce our proposed systematic methodology in section 3.1 that is used for automatic selection of hardware assertions that can be used to improve observability as well as reducing error detection latency. We will then define two quantitative metrics in section 3.1.6 that can be used to assess the quality of the selected assertions in detecting bit-flips in flip-flops. A heuristic algorithm is the integral part of the methodology which is fully detailed for both coverage metrics in section 3.2. We will then provide experimental results that discuss our findings regarding the introduced coverage metrics as well as the resulting area overhead in section 3.3. Finally, the chapter is closed by providing concluding remarks in section 3.4.

3.1 Proposed Methodology

Before elaborating on the main steps in our methodology, we first motivate the idea of using assertions for bit-flip detection.

3.1.1 Potentials of Assertions

There exist two error scenarios [140]: *silent errors* and *masked errors*. A silent error occurs when an error propagates to an observable point but is missed due to insufficient checking. On the other hand, a masked error is an error that is produced but masked out before reaching an observable point. Due to the inherent lack of real-time observability in circuit blocks that are deeply embedded into the design under validation, depending on the workload, most bit-flips will not manifest themselves at an observable output, despite the fact that their presence proves an underlying problem with the design. For example, some of our exploratory simulation experiments have shown that for the s38417 circuit (from the ISCAS89 benchmark set [8]) on average, only one out of the ten injected errors were observed at the primary outputs. Moreover, even if these intermittent errors do propagate to primary outputs, checking them against a pre-computed "golden response" (complete and comprehensive response of the circuit to all enumerations of the input space) is infeasible in post-silicon validation. This is because of the huge volume of clock cycles that are applied to silicon prototypes, which makes pre-computation of "golden response" impractical in simulation environments. Another concern that arises when bit-flips are not detected soon after they occur is because failing experiments, which are caused by bit-flips, are not easily reproducible due to the electrical phenomena that cause them (e.g., unique temperature and power supply noise). Hence, one must ensure that the trace signals that are collected on embedded trace buffers (which are also constrained in size)[141, 142] are most relevant to the error that has occurred. This recorded information is critical during root-causing [143] and, to ensure that meaningful information is analyzed, the error detection latency must not exceed the depth of the trace buffers. Therefore, considering the goal of low error detection latency, employing assertions during post-silicon validation has been motivated by the following points:

- Assertions can perform property checking without requiring a "golden response";
- Techniques, such as [134], have been proposed for mapping assertions to hardware, thus making them suitable candidates for post-silicon validation;
- Traditionally, assertions have been carefully crafted by verification engineers before being deployed. The drawback of this approach was the omission of non-obvious assertions that were beyond the understanding of verification engineers. However, recent research works, have explored automatic assertion generation using different methods ranging from static or dynamic analysis as in [144, 145, 2] or by employing data mining as in [3] and [146].

Although the original goal of automated assertion generation is to aid pre-silicon verification, e.g., when the design implementation changes iteratively or design blocks are reused in different environments, we recognize the potential of these discovered assertions for post-silicon validation. Therefore we build our methodology (illustrated in Figure 3.1) by leveraging the recent advancements in both assertion discovery [3] and their hardware mapping [134]. A key observation is that bit-flips, unlike functional errors, are related to the design netlist, which facilitates both a concise definition of the error space to be covered, as well as the development of a method that does not rely on design functionality but rather on its structure. This, in turn, facilitates automation in a manner that resembles common tasks in the electronic design flow, such as, for example, logic synthesis, place and route, or automatic test pattern generation. Nonetheless, to make the methodology practically feasible, one has to account for unique hardware constraints. Therefore, the huge number of assertions that are discovered during the pre-silicon phase need to be consciously selected before being mapped to hardware, which is an important novel aspect of our work.

3.1.2 Automatic Assertion Generation

As illustrated in Figure 3.1, the first step in our methodology is to discover assertions for a given design. Although one can develop assertions manually, in order to enable an automated methodology, it is necessary to rely on tools that can generate non-obvious assertions that cut across time cycles automatically which are also inter-modular. In other words, assertions must be automatically extracted from the given design (either netlist or RTL code) irrespective of the circuit's functionality. It is important to note that the assertions we use in this work are implementation dependent which means that we check the circuit's implementation against its silicon prototype. Hence, we are not concerned about assertions that are based on, for example, the high-level specifications of the circuit.
As discussed in chapter 2 section 2.3, there are two commonly used languages for writing assertions: Property Specification Language (PSL) and System Verilog Assertion (SVA). An example of an SVA assertion is shown below:

```
assr1:  ((x == 1) && (y == 0) |=> ##2 (a == 0))
```

where a is the destination signal and x and y can be flip-flops, primary inputs or internal nets. The ## indicates that the logic value of a needs to be evaluated on the second evaluation point (clock edge etc.) after the enabling condition of the assertion (antecedent side) has been satisfied.

Most of the available tools for automatic assertion generation are customizable [3, 2], thus meaning that the user can choose the destination flip-flops. Since the objective is to detect bit-flips that occur in flip-flops, assertions that target flip-flops as destination signals are deemed to be the superior ones to be added to the discovered assertion pool. For instance, in the assertion statement above, if a is a flip-flop and a bit-flip occurs in the circuit changing its value from 0 to 1, and all other conditions hold, then this assertion would be violated. If the status of the assertions is monitored during circuit's operations, then this bit-flip can be detected as soon as the assertion is fired. We call a flip-flop as a *potentially covered* flip-flop if at least one assertion fires for at least one bit-flip that affects it. We will come back to the topic of coverage in section 3.1.6.

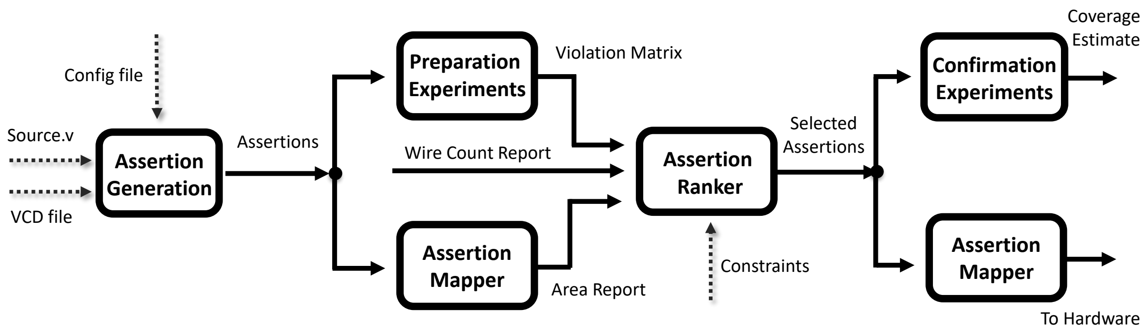


Figure 3.1: Tool flow for selecting the most suitable assertions to embed on-chip under wire constraints

3.1.3 Preparation Experiments

Our exploratory experiments (on the ISCAS89 circuits [8]) indicate that for a design block with one to two thousand flip-flops an assertion discovery tool can produce more than twenty thousand assertions. Mapping all these assertions to hardware is obviously impractical due to both *area* and *wiring* constraints. Consequently, assertions need to be weighted and a subset of them must be shortlisted as *selected assertions* for hardware mapping.

Since our objective is to improve the number of flip-flops that are potentially covered when bit-flips occur, assertions that are more likely to violate within a predefined time window in response to bit-flips are preferred over the other ones. For example, if, during a bit-flip injection experiment, *assr i* detects 12 different bit-flips and *assr j* detects 5 different bit-flips, but 4 of these bit-flips that are detected by *assr j* are also detected by *assr i*, then it would be logical to select *assr i* as the selected assertion and dismiss *assr j*. There are many other factors, other than the violation count, that need to be taken into account and they will be all elaborated in section 3.2.

Since the potential of each assertion to detect specific bit-flips needs to be worked out such that ranking will be done based on this potential, following to assertions discovery, *preparation experiments* are carried out in order to determine the violation count for each assertion when bit-flips are randomly, but uniformly, injected in all the flip-flops. As shown in Figure 3.2, the following steps are carried out to perform the preparation experiments:

1. Initially, each flip-flop is instrumented with a 2-to-1 mux and an inverter to facilitate error injections (bit-flips). The select signal of the mux determines

when and where the bit-flip should occur. In addition to instrumenting flip-flops, all the discovered assertions are added to the design. Note that functional simulators do support high-level assertions in PSL and SVA format. Hence, there is no need to synthesize assertions to their hardware equivalent in this step.

2. For each simulation, we load the circuit in a random state. We wait for a user-defined time (e.g., 10 clock cycles), during which we monitor assertions to make sure no violation happens. In case there is an assertion violation, it suggests that the initial state is illegal in which case the simulation is dismissed and a new initial state is tried.

3. If there is no violation after that user-defined period, we will target one flip-flop at a time, inject bit-flips and simulate the design using random input stimuli. In each simulation, bit-flips are injected at k random times and the circuit is simulated for a user-defined time (e.g., 256 clock cycles) after error injection; during which assertions are monitored and their violation is recorded, as illustrated in Figure 3.2. As a result, if the circuit has m flip-flops, the total number of simulations that have to be performed will be $m \times k$.

4. Finally, by combining the violation reports for each simulation, a $m \times n$ matrix is created where m is the total number of flip-flops and n is the total number of assertions. Each entry in this *Violation Matrix* represents the total violation count of an assertion for a specific flip-flop in all simulations. For instance, entry (1,2) in Figure 3.2 means that *assr1* has been violated 12 times for all the errors injected in *flip-flop 1*.

It is important to note that the violation matrix captures the error space of bit-flips that can be detected by the assertions that were assessed during the preparation experiments. Due to the randomness in preparation experiments, which is needed to account for the random occurrence of bit-flips on silicon prototypes, an assertion that fired for one bit-flip injection might not be violated for another bit-flip injection in the same flip-flop; this can happen due to multiple reasons. For instance, it is likely that the circuit is in a different state for which the signals that imply the assertion have different value. Also, it is extremely probable that the effect of the bit-flip might not propagate to the assertion checker due to a different input sequence. This is the reason for referring to the flip-flops from the violation matrix, for which at least one assertion has fired during the preparation experiments, as *potentially covered*. It is

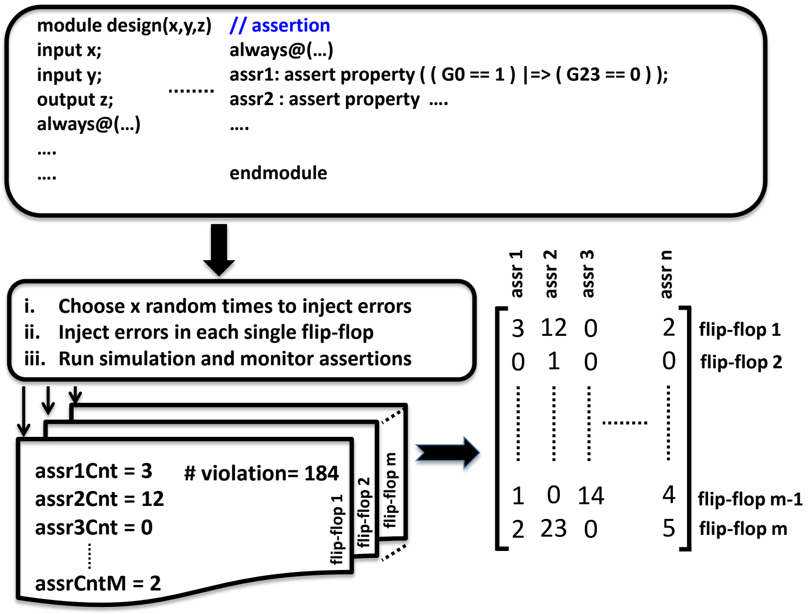


Figure 3.2: Preparation Experiments illustrating steps towards creation of the Violation Matrix. Each entry in this violation matrix shows the total number of violations of the assertion (from the corresponding column) when the bit-flip was injected in the respective flip-flop (identified by the row).

up to the user of the methodology to exclude or include flip-flops of interest for the bit-flip injection experiments. Hence, the number of rows in the violation matrix is upper-bounded by the number of flip-flops in the design and the number of columns, i.e., the number of assertions to be considered by the preparation experiments, can be bounded based on, for example, the available computational resources. It should also be noted that the quality of the violation matrix is central to the accuracy of the assertion ranking algorithm. The more simulations we run during the preparation experiments, the more revealing is the information in the violation matrix.

3.1.4 Mapping Assertions to Hardware

Traditionally, assertions have been developed for verification and are composed of logical and temporal operators and regular expressions. These statements can be added to the source code in pre-silicon verification to monitor errors using functional simulators. However, for using them in post-silicon validation, they must be mapped into hardware in order to do on-line property checking. Both PSL and SVA assertions are not synthesizable by default. However, as mentioned before, there are tools such as [134] that can accomplish assertion synthesis. Once assertions are discovered, assertion mapping can be done simultaneously with the preparation experiments. This will provide accurate area estimates for each assertion, which are needed by the ranking algorithm, as it can be seen in Figure 3.1.

3.1.5 Assertion Ranking

Due to the imposed area and wiring constraints, implementing all assertions on-chip is impractical. As an example, our exploratory experiments on s35932 circuit [8] has shown that if all the discovered assertions are added to the circuit, the area associated with them can easily exceed 20 times that of the area of the circuit itself. Therefore, the large pool of available assertions must be assessed and subsequently, only a subset of them are chosen and marked as selected assertions to be embedded into hardware. In this work, by having established a relationship between assertions and bit-flips that they might detect in section 3.1.3, we tune the assertion ranker to target two different objectives. At first, as it will be described in section 3.2.1, we focus on maximizing bit-flip detection. Toward this goal, we elaborate an algorithm that focuses on selecting a subset of assertions that will maximize the number of bit-flips that are detected through violation of at least one of the embedded assertions. Afterwards, we propose another algorithm that focuses on maximizing the number of flip-flops that are potentially covered (as defined earlier), under user-provided constraints. The ranking algorithms, which will be detailed in section 3.2, use the violation matrix, area estimates, the wire count report and user-specified constraints. The required steps to prepare the violation matrix were thoroughly explained in section 3.1.3; likewise, area estimates for assertions are obtained as explained in section 3.1.4. The wire count report can be directly extracted from the assertions pool by counting the distinct number of wires that comprise each assertion statement. Finally, the constraints are provided by the user. In our current implementation, we provide the wire count as the constraint.

3.1.6 Confirmation Experiments

The last step in our methodology is to run confirmation experiments. During confirmation experiments, the circuit is instrumented by the assertions that have been selected by the specific algorithm in assertion ranker. The circuit is then simulated using random input stimuli during which one error is injected at a time (injections will be uniformly distributed across all the flip-flops throughout the entire duration of the confirmation experiments) and the assertions violations are recorded. In the ideal case, whenever a bit-flip is injected into an arbitrary flip-flop, a violation will be detected if at least one of the selected assertions that are embedded into the design fires in response to the bit-flip. However, considering the random occurrence of bit-flips, it can happen that some bit-flip injections will not cause a violation despite the fact that based on the violation matrix, more than one of the selected assertions were expected to fire. Furthermore, though it is less likely, it might also happen that some of the selected assertions, that were not identified during the preparation experiments to be related to a particular flip-flop, will fire during the confirmation experiments when a bit-flip is injected in the respective flip-flop. In any case, if at least one bit-flip in a flip-flop causes a violation in at least one of the embedded assertions, that flip-flop is potentially covered by that assertion. At this point, we define two different metrics to evaluate the effectiveness of our algorithm, which can also be used as an internal feedback to the entire methodology; *bit-flip coverage estimate* and *flip-flop coverage estimate*.

Definition 1. *Bit-flip coverage estimate* is defined as the ratio between the total number of bit-flips that are detected by the selected embedded assertions and the total number of bit-flips that are injected during the confirmation experiments.

Definition 2. *Flip-flop coverage estimate* is defined as the ratio between the total number of flip-flops for which there exists at least one assertion that has detected at least one of the bit-flips that have been injected in that flip-flop over the total number of flip-flops.

As an example, in the hypothetical circuit of Figure 3.3 with three flip-flops, 4 bit-flips are injected in each flip-flop. Consequently, according to the definitions above, the bit-flip coverage estimate is 42% whereas the flip-flop coverage estimate is 67%.

The distinction between these two metrics stems from the fact that due to the reasons that were elaborated in section 3.1.3, it does happen that an injection in a flip-flop causes all the associated assertions for that flip-flop (or other assertions if the error propagates to other flip-flops) to fire while another error in the same flip-flop might end up undetected. Since post-silicon validation sessions commonly run for extensively long durations, it is our position that *flip-flop coverage estimate* is of higher importance. This is because if even a single bit-flip is detected in a flip-flop over a long validation experiment (hours of real-time execution), this will highlight the presence of a subtle electrically-induced error within the close proximity of the detection point that needs to be corrected before committing to high-volume manufacturing.

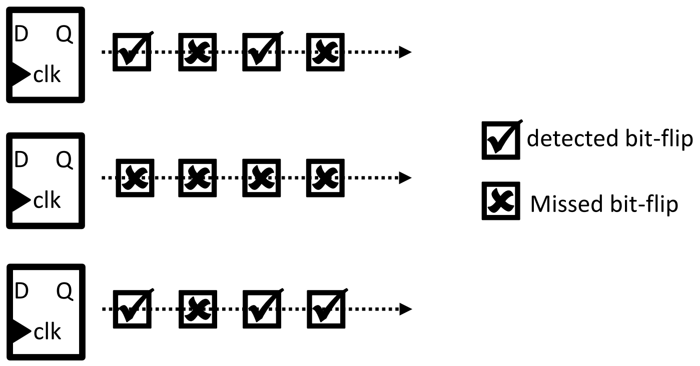


Figure 3.3: Hypothetical example for representing the difference between bit-flip and flip-flop coverage estimates.

It is important to note why both coverage metrics are labeled as *estimates*. Bit-flips, unlike hard defects (modeled as, for example, stuck-at faults in the logic domain) occur only in some unanticipated clock cycles, dependent on the logic state of the circuit, the workload and the electrical state (e.g., voltage supply and/or droop). One cannot guarantee a bit-flip to be easily reproducible, nonetheless, so long as the validation sequences are long (as it is the case for post-silicon validation) it is expected that the underlying electrical problem will manifest itself as a bit-flip in the logic domain. It is needless to say that our confirmation experiments are limited in duration (when compared to post-silicon validation experiments), nonetheless if an injected bit-flip is detected during the confirmation experiments by the subset of selected assertions, we *estimate* that a post-silicon validation experiment would also detect it because of the huge volume of additional clock cycles that are applied in post-silicon validation (approximately six orders of magnitude more than simulation).

Finally, both these metrics can reveal important characteristics and features about the effectiveness of the embedded DFD logic and can also provide important feedback to different steps of the proposed methodology. For example, the original pool of assertions might need to be expanded if the confirmation experiments indicate that, for some flip-flops, no bit-flips were detected; it is also possible that an insufficient number of preparation experiments have been carried out, in which case the confirmation experiments will indicate that the quality of the violation matrix needs to be improved. Moreover, these *coverage estimates* can also serve as proofs of due diligence when the subset of the selected assertions that were used on the silicon prototype exhibit no failure after extensively long post-silicon validation experiments; in which case, the confidence to commit to high-volume of manufacturing is substantiated by the values captured by the coverage estimates. As a final note, we would like to point out that the coverage metrics we have introduced are based on the violation of assertions in hardware that have been discovered based on the design implementation and are of a different concern than researches based on high-level specifications [147, 148].

3.2 Ranking Algorithm

As emphasized in the previous section, mapping all the discovered assertions from the assertion generation step to hardware for the purpose of low-latency bit-flip detection is impractical due to area and wire usage overheads. Hence, in this section, we provide algorithms that are geared towards maximizing the two coverage metrics that were introduced in section 3.1.6; *bit-flip coverage estimate* and *flip-flop coverage estimate.*

3.2.1 Bit-flip coverage estimate maximization

As summarized in section 3.1.5, we present a novel heuristic algorithm to select a subset of assertions by accounting for a wire usage constraint and with the objective of maximizing the number of bit-flips that are detected. In other words, based on the definition given in section 3.1.6, the proposed algorithm will work on maximizing the bit-flip coverage estimate. A bit-flip is considered as detected if, after its occurrence, at least one of the assertions that are embedded in the design exhibits a violation, in which case the bit-flip is detected. Note that, according to this definition, there is no benefit on having multiple assertions violating for a single bit-flip. Figure 3.4 shows the flow for the proposed algorithm.

- In order to prioritize one assertion over another, a metric is defined for the

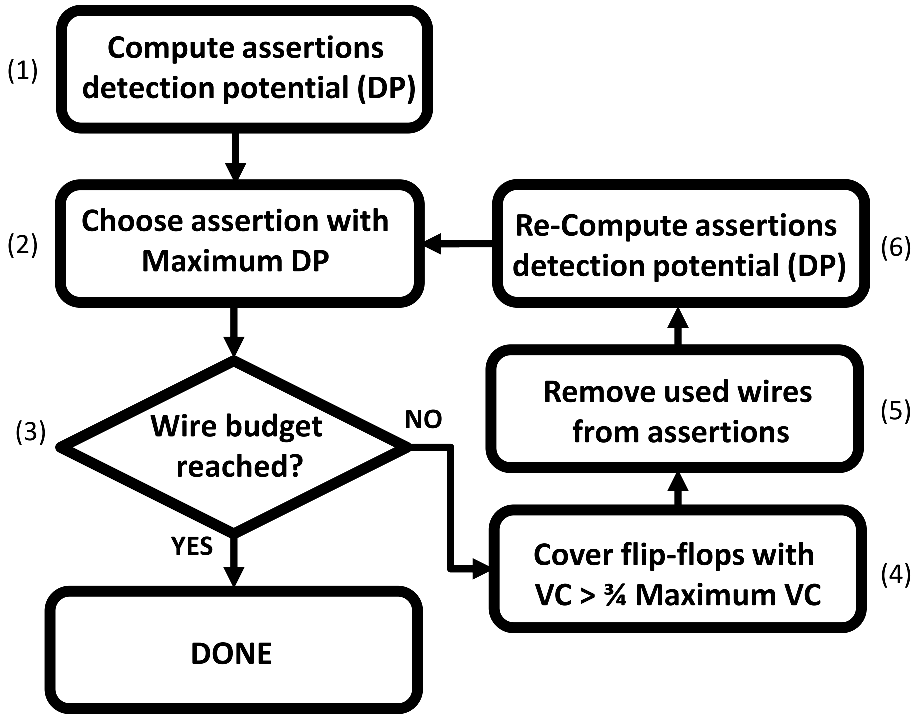

(1)
Compute assertions detection potential (DP)
(2)
Choose assertion with Maximum DP
(3)
Wire budget reached?
NO
YES
DONE
(4)
Cover flip-flops with VC > ¾ Maximum VC
(5)
Remove used wires from assertions
(6)
Re-Compute assertions detection potential (DP)

Figure 3.4: Flow for the heuristic algorithm that is geared towards maximizing bit-flip coverage estimate. Note that in step 4, VC represents the assertions violation count for a particular flip-flop in the violation matrix.

purpose of one-to-one comparison (step 1). We define the *detection potential* (DP) of each assertion as:

$$Assr(i)_{DP} = \frac{TotalViolation}{WireCnt}$$

where `TotalViolation` is cumulative sum of violation counts for *assr(i)* in all experiments during preparation experiments and `WireCnt` is the number of distinct wires used in *assr(i)*. Intuitively, assertions with a high violation count and low wire count are prioritized.

- Once the *detection potential* is computed for all the assertions, the next step would be to select the one with highest DP (step 2).

- After each selection, it is mandatory to check if we have reached the available wire budget (step 3). If the wire budget is reached, the algorithm should stop. Otherwise, we move onto the next step.
- If we are still within the wire budget, then the next step is to find all the flip-flops whose violation count are greater than or equal to three quarters of the maximum violation count for that flip-flop. This is because, in bit-flip coverage, all effort must be made to make sure that as many bit-flips as possible are detected. Hence, once an assertion is chosen, by considering its violation count for all the flip-flops, one can conclude that the likelihood of that assertion detecting bit-flips for a flip-flop whose current violation count is three quarters of the maximum violation count in the relative row of the violation matrix for that flip-flop is sufficiently high to consider bit-flips in that particular flip-flop as covered (step 4). Note that the threshold of three quarters of the maximum violation count is based on our empirical observations in the use-cases that we will discuss in section 3.3 and can be adjusted by the user of the methodology.
- At this point, it is sensible to remove wires that have already been used by the selected assertions from the remaining assertions, so that the remaining assertions only contain wires that are distinct from the set of wires for all the assertions that were selected so far. For example, assume assertion i has five wires and that two of those wires also belong to assertion j; after selecting assertion j, the cost of selecting assertion i will be three wires instead of five (step 5).

- Finally, the detection potential is recomputed based on the new wire counts for each assertion (step 6) and the next assertion with the largest detection potential is chosen.

3.2.2 Flip-flop coverage estimate maximization

Bit-flip coverage estimate is an important measure on evaluating the effectiveness of assertions that have been selected by the ranking algorithm. However, as stated before, since post-silicon validation sessions run for extensively long durations, flip-flop coverage estimate (definition given in section 3.1.6) is of a higher importance during post-silicon validation. This is due to the fact that, if there is at least one failing experiment among several debug sessions, it suggests a subtle underlying problem in the design that must be diagnosed and repaired before committing to high-volume manufacturing. In other words, if the root cause of the electrical problem produces several bit-flips over a long validation experiment (hours of real-time execution), if at least one of these errors can be detected by the embedded assertions, it would be beneficial to point out an existing problem in the specific flip-flop or its vicinity. Hence, in this section, we propose a novel algorithm that, unlike the algorithm in section 3.2.1, works on selecting a set of assertions to maximize *flip-flop coverage estimate* by taking into account the wire usage constraint. The objective is to maximize the potential coverage for each flip-flop, while at the same time the area should be contained.

1. **Algorithm**

   The pseudo-code for this algorithm is given in Algorithm 1. Before we elaborate on the main steps of the algorithm, as in the case of bit-flip coverage estimate algorithm, we should define a metric that is used in order to weight assertions for the purpose of one-to-one comparisons. The *Detection Potential (DP)* that was introduced in section 3.2.1 is further extended to take into account the parameters that also contribute to the decision being made by the algorithm on selecting the most appropriate assertion. Hence:

$$Assr(i)_{DP} = \frac{(\alpha \times FCov + TotalViolation)}{(\beta \times WireCnt) + Area} \tag{3.1}$$

The terms in the detection potential and also the different steps in the algorithm are described as follows:

- Initialization: At the beginning, the backbone data structure that is the basis of violation matrix is constructed by reading the associated data from experimental results and wire and area report files (step 1).
- TotalViolation: After creation of the violation matrix, the sum of violation counts of each assertion for each flip-flop is calculated (column sum). The violation count of each assertion for a specific flip-flop is the total number of times that a specific assertion has been violated when a bit-flip has been injected in that flip-flop (step 2).
- FlopCov: This stands for the flip-flop coverage of an assertion, that is, the total number of flip-flops for which the associated violation count entry in the violation matrix is greater than zero (step 3).
- In order to bring `Area` and `TotalViolation` to the same scale, the respective values for these two attributes are scaled linearly in such way that in each iteration, the minimum and maximum values of these attributes

```
input  : Violation Matrix, Wire and Area reports
output: Candidate Assertions
initialization;                                                        // step 1
while Used Wires $\leq$ wire Budget do
    foreach assertion in assertion list do
        find TotalViolation;                                           // step 2
        find FCov;                                                     // step 3
        scale Area and TotalViolation;                                 // step 4
        find $\alpha$, $\beta$;                                        // step 5
        find DP;                                                       // step 6
    end
    Possible Candidates = Assertions with DP within 1% of Max DP;      // step 7
    foreach $assertion_i$ in Possible Candidates do
        find $\sigma_i$;                                               // step 8
        $DP_i = \frac{DP_i}{\sigma_i}$
    end
    select selectedAssr;                                               // step 9
    usedWire += $selectedAssr_{wire}$;                                 // step 10
    usedArea += $selectedAssr_{area}$;                                 // step 11
    foreach $FF_j$ in FlipFlops do
        if $selectedAssr_{vc}$ of $FF_j > 0$ then
            cover $FF_j$;                                              // step 12
        end
    end
    foreach $assertion_i$ in assertion list do
        update $assertion_i$ wires;                                    // step 13
    end
    if AllFlopsCovered then Break                                      // step 14;
end
```

Algorithm 1: Ranking Algorithm for maximizing flip-flop coverage estimate.

are measured and the associated area and total violation count of each assertion is scaled accordingly in a linear manner (step 4).

- $\underline{\alpha, \beta}$: These coefficients are used to bring different terms in the nominator and denominator of the detection potential formula to the same scale, so that one term is not accidentally given more importance than necessary. However, if the user wants to deliberately adjust the importance of one specific term (e.g. FlopCov) in the detection potential equation, it can be done by multiplying that term further with another coefficient. For the sake of clarity, we have omitted that extra coefficient. The α and β scale factors are defined as:

$$\alpha = \frac{T_{avg}}{F_{avg}} \quad \beta = \frac{A_{avg}}{W_{avg}}$$

  where T_{avg} is the average of the total violation counts for all assertions, F_{avg} is the average of the total flip-flop coverage for all assertions and A_{avg} and W_{avg} stand for the area and wire count average of all assertions respectively. As it will be clarified later, all these coefficients are calculated based on the remaining assertions and their updated wire counts in each iteration of the algorithm (step 5).

- Once all the prior steps are done, the detection potential for each assertion can be calculated according the Eq. 3.1 (step 6). Note that the wire count of each assertion has been determined in step 1.

- At this point, all assertions for which their associated DP is within 1% to the maximum DP are marked as Possible Candidates and are taken for future evaluations (step 7). The selected assertion should not have a significant discrepancy in violation counts for different flip-flops, since it will likely have a higher potential to detect bit-flips for all the flip-flops that it is related to. Therefore, for all assertions in the Possible Candidate list, the violation count standard deviation (σ) of those assertions must be calculated and their respective DP is divided by this standard deviation (step 8). The lower the standard deviation is, the larger the DP will be.

- As soon as the detection potential (DP) values of all assertions are available, the assertion with the highest DP is selected as the selected assertion (step 9), and the wires that are part of this assertion statement are added to the used wire list (step 10). Equally, the associated area of the assertion is also added to the total area usage (step 11).
- Following to selection of the selected assertion, all the flip-flops that are potentially covered by it are marked and taken out of further consideration. A flip-flop is potentially covered by an assertion if the corresponding entry of the violation matrix for that flip-flop has a non-zero violation count (step 12).
- The algorithm continues by carrying out an important task that is to update the wire count for each assertion. This is crucial because, in the following iterations when the DP of each assertion is to be determined, the wires that have already been used should not be taken into consideration. For example, during evaluation of assertions DP, an assertion that initially had five wires but three of its wires are already in the used wire list will be treated as an assertion with only two wires (step 13).
- Finally, before moving to the next iteration, though unlikely, it is sensible

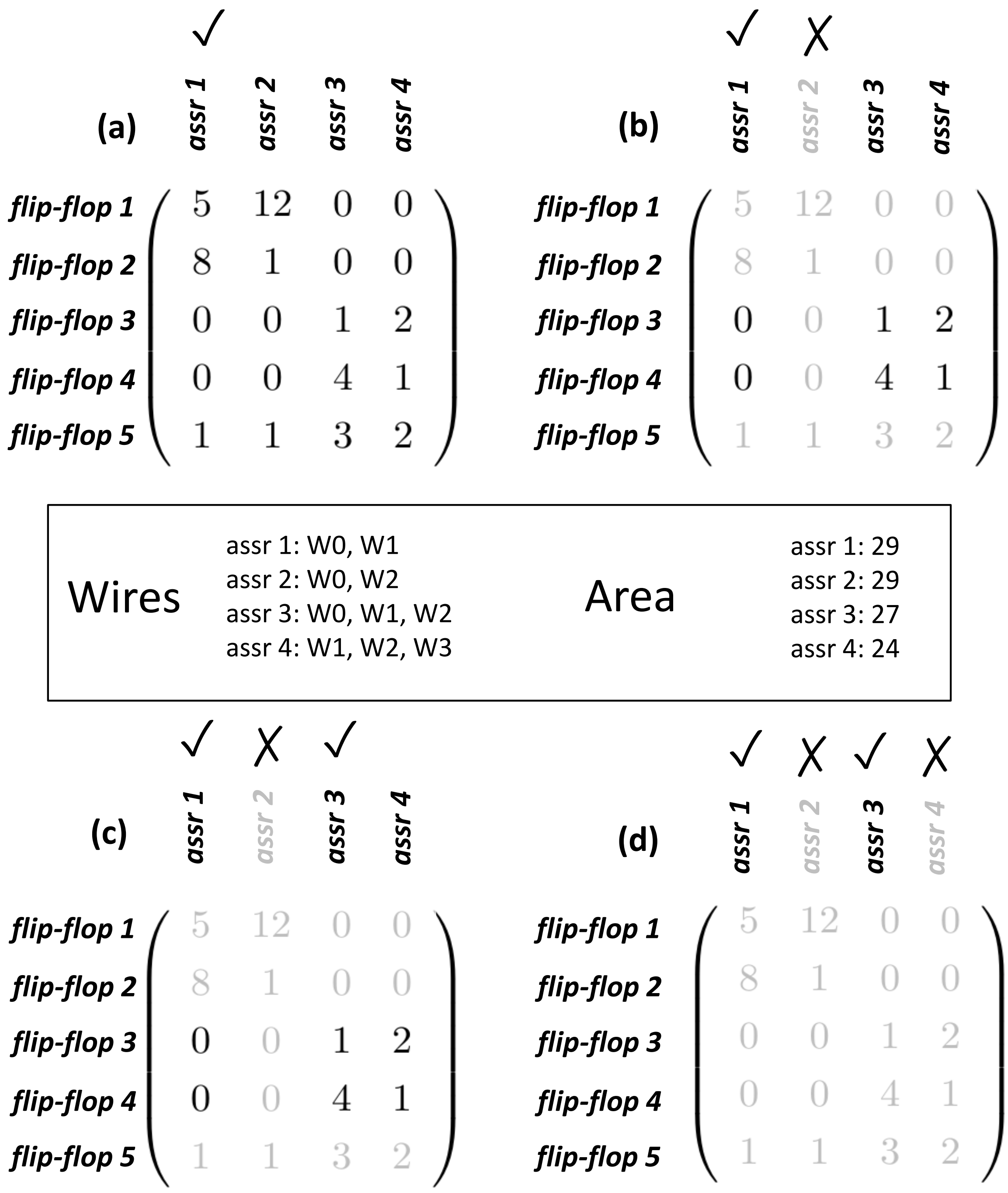
to check if all the flip-flops have been potentially covered by the already selected assertions so that there is no need to select more assertions (step 14).

In the following section, all steps of the algorithm are reviewed by the aid of an example.

2. **Example** The example in Figure 3.5 will illustrate the process of assertion selection in each iteration in the proposed algorithm. Initially, the detection potential of each assertion is to be calculated according to formula in Eq. 3.1. Hence:

$$\begin{aligned} assr(1)_{DP} &= \frac{(19.5 \times 3) + 100}{(26.04 \times 2) + 100} = 1.04 \\ assr(2)_{DP} &= \frac{(19.5 \times 3) + 100}{(26.04 \times 2) + 100} = 1.04 \\ assr(3)_{DP} &= \frac{(19.5 \times 3) + 33}{(26.04 \times 3) + 59.4} = 0.66 \\ assr(4)_{DP} &= \frac{(19.5 \times 3) + 1}{(26.04 \times 3) + 1} = 0.75 \end{aligned}$$

Note that α and β have been calculated based on the scaled values of TotalViolation and Area. As it can be seen, the maximum detection potential is 1.04 and is associated with both assertion 1 and assertion 2. This is because both assertions have similar `TotalViolation`, `FCov`, `WireCnt` and `Area`. Referring to the ranking algorithm (Algorithm 1), the standard deviations for both assertions have to be calculated and their detection potential must be divided by the respective standard deviation. After the division, the assertion with highest DP is picked as the selected assertion and the other one is dismissed from further



$$\begin{array}{c|cccc} & \text{assr 1} & \text{assr 2} & \text{assr 3} & \text{assr 4} \\ \hline \text{flip-flop 1} & 5 & 12 & 0 & 0 \\ \text{flip-flop 2} & 8 & 1 & 0 & 0 \\ \text{flip-flop 3} & 0 & 0 & 1 & 2 \\ \text{flip-flop 4} & 0 & 0 & 4 & 1 \\ \text{flip-flop 5} & 1 & 1 & 3 & 2 \end{array}$$

Wires		Area	
assr 1	W0, W1	assr 1	29
assr 2	W0, W2	assr 2	29
assr 3	W0, W1, W2	assr 3	27
assr 4	W1, W2, W3	assr 4	24

Figure 3.5: An example showing the different steps on choosing a sub-optimal set of assertions to maximize flip-flop coverage estimate.

evaluation. For our example, the DP values become:

$$assr(1)_{DP}' = \frac{1.04}{2.867} = 0.362$$
$$assr(2)_{DP}' = \frac{1.04}{5.185} = 0.200$$

Hence, assertion 1 which has a higher DP is selected as the first selected assertion. Taking standard deviation into consideration ensures avoiding an assertion that covers a dominant flip-flop (that contributes most to its violation count) and potentially misses the other flip-flops. Subsequent to selecting assertion 1, its wires then have to be added to the used wire list. Moreover, all flip-flops that are potentially covered by this assertion are also marked and taken out from further consideration (Fig. 3.5, part b). Assuming that the wire budget is 6, and also the fact that not all the flip-flops have been potentially covered, the algorithm goes to the second iteration. During the second iteration, all the values in the DP formula must be updated in such way that attributes (Wires, FCov, etc.) for flip-flops that have been covered already should not be counted. Therefore, the detection potentials for the second iteration will be:

$$assr(3)_{DP} = \frac{(25.25 \times 2) + 100}{(33.67 \times 1) + 1} = 4.34$$
$$assr(4)_{DP} = \frac{(25.25 \times 2) + 1}{(33.67 \times 2) + 100} = 0.307$$

Since the detection potential of assertion 4 is not within 1% of the maximum detection potential (assertion 3), there is no need to calculate the standard deviation. Hence, assertion 3 is selected as the second selected assertion, its wires are added to the used wire list and also the flip-flops that are covered by it are marked as potentially covered. As it can be seen in Figure 3.5, part (d), by selecting assertion 3, all the remaining flip-flops in the design are potentially covered, hence the algorithms stops.

3.3 Experimental Results

In this section, we provide and discuss the experimental results. Our methodology, together with both ranking algorithms, have been implemented on an Intel Core i7 machine with 32GB of RAM using GCC 4.8.2. For assertion discovery, we have used GoldMine, an automatic assertion generation tool that uses data mining and formal verification. GoldMine first generates likely invariants based on the simulation trace of the given circuit. These likely invariants are then passed to Cadence Incisive formal verifier which only selects those invariants that are true assertions (i.e. properties that hold indefinitely). GoldMine offers several modes of operation and the detailed discussion on its different modes and configurations are out of the scope of this work; the interested reader is referred to [3]. Results from a combination of mining engines available in GoldMine have been used for gathering assertions in our test cases; though we should note that decision forest and coverage mode engines have been used more often than the other engines. With regard to the input stimuli for assertion discovery, we have used both random inputs (default of GoldMine) and deterministically generated value change dump (VCD) files obtained through Validation Vector Generator tool from Virginia Tech [149, 150]. As for the input stimuli for experiments, both in preparation experiments and confirmation experiments, we have used random inputs throughout our experiments to reflect the random occurrence of bit-flips in post-silicon environments. In pursuance of obtaining the hardware equivalent circuit of the assertions (to estimate their area), all high-level SVA assertions have been added to the original source code and passed to MBAC[134] to obtain their Verilog code. Once the synthesizable Verilog model for all the assertions is produced, their respective area estimate is determined using Synopsys Design Compiler (based on generic implementation libraries). After creating the *violation matrix* (the output of preparation experiments explained in section 3.1.3) and extracting area estimates and wire counts of assertions, the ranking algorithm of choice will select a subset of assertions depending on whether the objective is to maximize bit-flip coverage estimate or flip-flop coverage estimate. Although users can choose constraints specific to their hardware environment, in our current implementation we have used the number of used wires by assertion propositions as the constraint. As for the test cases, we have reported results on the three largest ISCAS89 benchmark circuits[8], the *quantizer* block (y quantizer) of the hardware implementation of the JPEG encoder and also the hardware implementation of the Reed Solomon decoder (without embedded RAMs). The latter two open-source benchmark circuits are available on OpenCores.org. Since the ranking algorithm is constrained with the number of used wires that is varied with respect to the total number of flip-flops in the design, the number of flip-flops in the respective benchmark circuits is given in Table 3.1.

Before the results of coverage metrics are discussed, as motivated in section

Circuit	No. of flip-flops
ISCAS s35932	1728
ISCAS s38417	1636
ISCAS s38584	1452
JPEG y quantizer	2823
Reed Solomon (R.S.) decoder	2885

Table 3.1: Number of flip-flops in the benchmark circuits used in the experiments.

earlier, the key reason to use assertion checking for bit-flip detection is to minimize the error detection latency. Table 3.2 shows the number of errors that fall into different error detection latency windows. As it can be seen in this table, the vast majority of the bit-flips are detected in less than 10 clock cycles after their occurrence. An interesting topic to explore in future work is how one can trade-off the error detection latency, which is acceptable to be in the range of tens to hundreds of clock cycles in practice, for a lower area and wire usage investments for the bit-flip assertion checkers.

Circuit	<5 clock cycles	<10 clock cycles	>10 clock cycles
s35932	98.1%	99.4%	0.6%
s38417	96.3%	99.1%	0.9%
s38584	98.4%	99.4%	0.5%

Table 3.2: Evaluation of the error detection latency for the detected errors.

In the following sub-sections, we discuss the results for the algorithms introduced for the bit-flip coverage estimate and the flip-flop coverage estimate.

3.3.1 Bit-flip Coverage Estimate

In this section, we review the results for the case when the ranker is configured to maximize the *bit-flip coverage estimate*. The violation matrix, which is central to the ranker, has been created by injecting 50 bit-flips uniformly in each flip-flop in the design. Once the violation matrix is prepared, the detection potential (DP) of each assertion can be worked out based on the sum of the violation counts of that assertion for each flop divided by the unique number of wires that constitute that assertion, as explained in section3.2.1. The results are

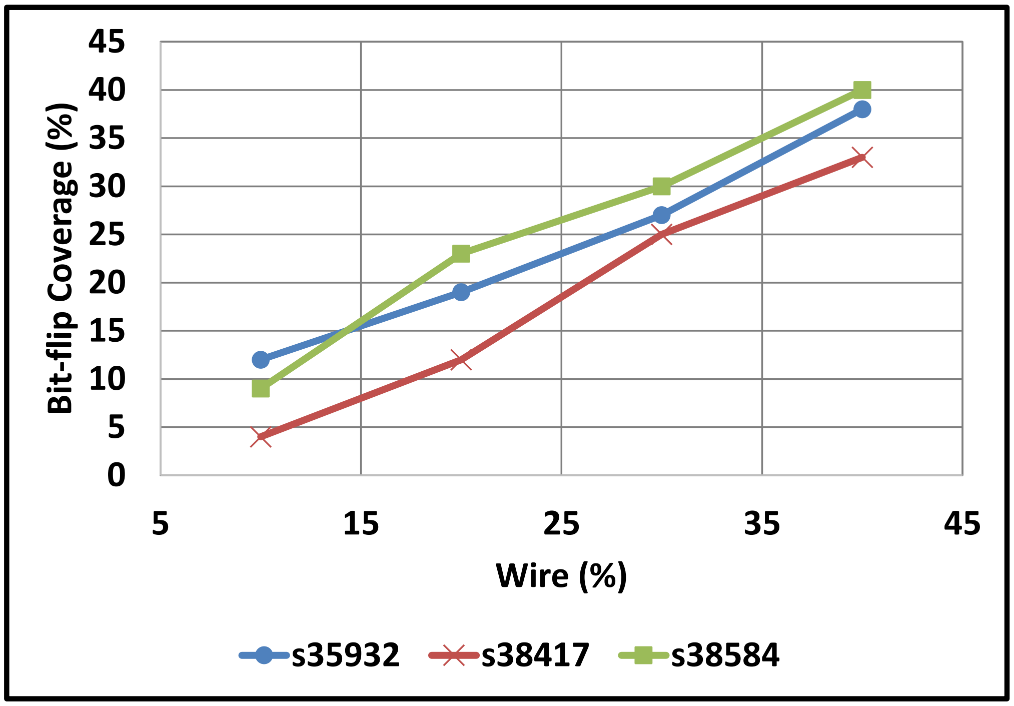


Figure 3.6: Bit-flip coverage estimate for the three largest ISCAS89 circuits [8].

shown in Figure 3.6. As it can be seen, by keeping the wire budget around 40% of the total number of flip-flops, the resulting bit-flip coverage estimate will also reside in the range of 35%. This is in part due to the fact that, some assertions detect multiple bit-flips and an assertion with 3 wires might actually detect 4 different bit-flips. In addition to this, in some subsequent increase in wire count budget in the ranker, the slope of the change in bit-flip coverage estimate is more than that of the previous step. This is because, as the wire budget increases, the number of wires that are shared between assertions also increases, which causes the ratio between the number of selected assertions and the number of wires used by these assertions to grow. This larger number of assertions per wire causes the bit-flip coverage to increase.

3.3.2 Flip-flop Coverage Estimate

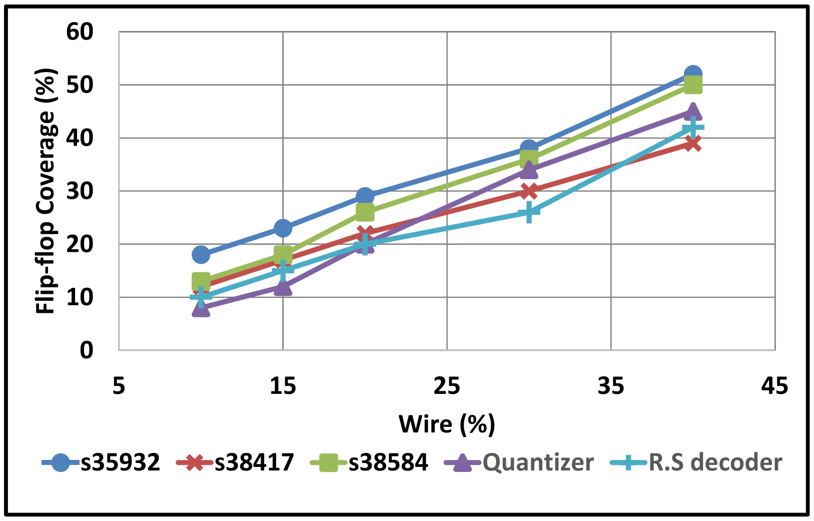
Although bit-flip coverage estimate is an important metric that can be used as an internal metric to assess the effectiveness and refine our methodology (e.g. initial pool of assertions), we believe that its practicality during post-silicon validation is limited. This is because post-silicon validation sessions run for long periods of time (days/months) with the main focus of detecting and localizing errors. In fact, a recent industrial study has shown that the time to detect an error is the dominant time during post-silicon validation [151]. If during a long validation experiment (on the order of days) at least one bit-flip (out of the potentially many that can affect a flip-flop) is detected by an embedded assertion, this information will unveil potential problems in the circuit and provide the critical info needed to root-cause the problem during the post-processing phase. It should be noted that the on-chip assertion which fires during a bit-flip can also be used as a trigger event to record data in a trace buffer [38, 141, 142], thus acquiring even more debug data for root-cause analysis. The *flip-flop coverage estimate* is a more appropriate metric to assess the benefit of the selected assertions for post-silicon validation because it reflects whether there exists at least one assertion that detects at least one of the bit-flips that occur in a flip-flop.

Figure 3.7a illustrates how flip-flop coverage estimate is changed as the number of wires that is provided to the ranker increases. For these experiments, the preparation experiments have been carried out using 50 of error injections per flip-flop and the confirmation experiments using 20 error injections. Since there exists an overlap between the nets that are used by different assertions, and also the fact that some assertions which have been used for this work detect multiple bit-flips, we can notice that, in general, the resulting flip-flop coverage estimate is greater than the number of wires that are used. In addition, in some occasions, when the wire count is increased, the slop of the change in flip-flop coverage increases (from 20% wire count onwards in s38584 and s35932) which is again the result of wire sharing between the assertions. It is worth mentioning that for circuit s38417 the coverage is lower. This is mainly due to the fact that the assertion pool used by the preparation experiments does not cover many flip-flops. This motivates future investigations into generating assertions that are focused exclusively toward bit-flip detection, which is expected to improve the quality of the violation matrix and indirectly the flip-flop coverage.

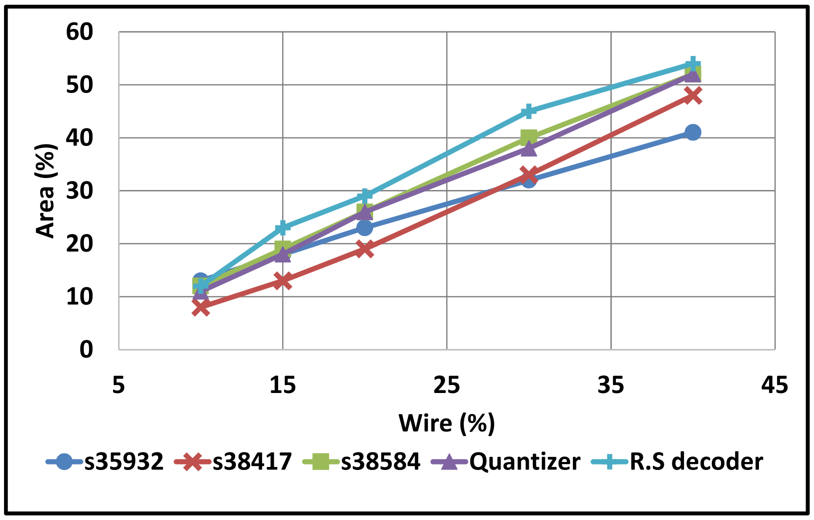
Figure 3.7b illustrates the area overhead of using embedded assertions when flip-flop coverage is to be targeted by the assertion ranking algorithm. The internal structure of the synthesized assertions is in such way that the logic blocks that are associated with each assertion are not shared between them. As a result, when the ranking algorithm constraint is relaxed to use a larger wire budget, the number of assertions that are selected grows, which directly increases the associated area overhead.

In the meantime, it is worth mentioning that all the results so far are based on the assumption that the likelihood of bit-flip occurrence is equal for all the flip-flops in the design and the ranking algorithm is configured to maximize flip-flop coverage of all the existing flip-flops. However, based on the study done in [152], the electrical bugs do not randomly affect all the flip-flops. It is claimed that the bit-flips in post-silicon validation are the result of excessive propagation delays that causes a delayed stabilization of the data at the inputs of flip-flops. The flip-flops that are mostly affected are timing-critical flip-flops and a bug is activated when the inputs of these flip-flops go through a transition. Motivated by these observations, we have adjusted our methodology and ranking algorithm to take into account only flip-flops that are timing-critical for the three largest ISCAS circuits [8].

As shown in Table 3.3, the flip-flop coverage estimate of the flip-flops on the critical path in s38584 has been measured when 50, 100 and 150 timing-critical flip-flops are taken into consideration. There is a saturation point in flip-flop coverage estimate for the cases when 50 and 100 flip-flops are targeted. This stems from the fact that for the remaining timing-critical flip-flops that have not



(a) Flip-flop coverage estimate when varying the wire count



(b) Area evaluation when varying the wire count

Figure 3.7: Analysis of flip-flop coverage estimate and the area overhead based on varying the number of wires, running the assertion ranker and carrying out the confirmation experiments. Wire count is the total percentage of wires used based on the number of flip-flops in the ISCAS89 benchmark circuit.

No. of Critical FFs	**Wire Count (%)**	**Area (%)**	**Flip-flop Coverage Estimate (%)**
50	2	2	40%
	4	4	76%
	6	5	86%
	8	5	86%
	10	5	86%
	15	5	86%
100	2	2	23%
	4	3	38%
	6	5	55%
	8	8	73%
	10	10	91%
	15	10	91%
150	2	2	15%
	4	3	26%
	6	4	37%
	8	6	48%
	10	8	57%
	15	14	88%

Table 3.3: Evaluation of flip-flop coverage estimate for flip-flops on critical path in ISCAS s38584 [8].

been detected, either there is no assertion in the original pool of assertions that can potentially cover these flip-flops (zero violation count for the row of those flip-flops in the violation matrix) or, though unlikely, the inputs that have been applied during bit-flip injections have not sensitized these assertions. Overall, it can be observed that the percentage of the critical flip-flops that are covered can be in the 90% range with wire counts of 15% or less of the total flip-flop count. We confirm that a similar trend exists for ISCAS s35932 when only the timing-critical flip-flops are taken into consideration. For ISCAS s38417 though, the flip-flop coverage of timing-critical flip-flops is poor due to the lack of availability of sufficient assertions for those flip-flops in the original pool of assertions. As mentioned earlier in this section, improving the original pool of assertions with assertions that are specifically focused on bit-flip detection, especially for flip-flops that are considered to be more susceptible to bit-flips, is a priority for future investigations.

3.3.3 Running Times

Finally, it is noteworthy to report the running times of the different steps in our flow. Results provided below in Table 3.4 are for the s38589 benchmark circuit. A similar trend has been empirically observed in all the use-cases that we have used to evaluate the proposed methodology.

Task	**Configuration**	**Time (hour)**
Assertion Generation	Miner (Single Core), Formal Verifier (Multi Core)	228
Preparation Experiments	Multi-core (4 instances)	63
Assertion Mapper (Synthesis)	Multi-core (4 instances)	1.8
Assertion Mapper (Area estimate)	Multi-core (4 instances)	9.7
Assertion Ranker	Single Core	0.32
Confirmation Experiments	Multi-core (4 instances)	11.5

Table 3.4: Evaluation of the running-time of different steps of the tool flow in Figure 3.1.

For assertion generation, a combination of results from the coverage miner engine and the decision forest miner engine of GoldMine [3] has been used with up to 25,000 input vectors with 6 to 10 counter examples fed back to the miner for input vector refinements. Mentor Graphics Questasim has been used for RTL simulation and MBAC [134] and Synopsys Design Compiler have been used for assertion synthesis and assertion area estimates respectively. For preparation experiments, a total of 28,365 assertions were assessed. Results given for the confirmation experiments refer to the worst case scenario where the ranker is configured with 40% wire count constraint, which results in 328 assertions being selected to be embedded in the design. Also, the results are for the case where 20 errors are randomly injected in each flip-flop of the design in both the preparation and confirmation experiments. As a final note, while considering the scalability of our proposed methodology, the author would like to add that it is a common practice in industry to validate designs on a block-by-block basis [153, 154]. The author believes that the size and complexity of the benchmark circuits that have been evaluated in this work are on par with the size of the sub-blocks in large industrial designs. Moreover, it is worth noting that both the preparation experiments and the confirmation experiments can be accelerated using FPGA-based emulation platforms as will be discussed in chapter 4.

3.4 Summary

In this chapter we have investigated the use of assertions in hardware for detecting electrically-induced errors that are manifested as bit-flips in flip-flops during post-silicon validation. As part of our methodology, we have designed algorithms to rank assertions based on their potential to detect bit-flips. We have introduced two quantitative metrics, the bit-flip and flip-flop coverage estimates, which can be used to assess the quality of the selected assertions.

As shown by our experimental results, a flip-flop coverage estimate of approx 50% is attainable when using a number of wires equal to 40% of the number of flip-flops in the design. If the total number of assertions is too large for practical applications, one can benefit from running multiple debug sessions during which assertions for a subset of flip-flops are mapped into an embedded programmable event-detector in a time-multiplexed fashion [6]. An alternative to time-multiplexing assertions, we have shown that the overhead is manageable if we monitor only the timing critical flip-flops.

To improve our methodology, the contributions in chapter 4 are concerned with improving the accuracy of the proposed methodology through hardware emulation. In addition, in chapter 5, we will present our own method for building a pool of assertions algorithmically, that are specifically focused on bit-flip detection and can be used as an input to the methodology presented in this chapter.

Chapter 4

Emulation Infrastructures for the Evaluation of Hardware Assertions

Chapter 3 detailed our methodology for automatic generation and selection of hardware assertions to facilitate bit-flip detection. As part of our methodology, we introduced bit-flip coverage estimate and flip-flop coverage estimate as two quantitative metrics for the assessment of the selected assertions to be embedded on-chip. Nonetheless, the main limitation of this methodology lies in its reliance on simulation-based experiments that capture a short snapshot of the design's behavior for assessing assertions potential for detecting bit-flips. This limits the number of error injections that can be performed, which indirectly affects the accuracy of selecting the embedded assertions checkers that will be committed to silicon for bit-flip detection. Since field-programmable gate-arrays (FPGA) emulators are capable of verifying logic designs at clock speeds of at least three orders of magnitude faster than a software simulator, they have been widely used to narrow the large gap between simulation and silicon speed [155, 156]. Moreover, FPGA-based emulation platforms have recently been employed for various post-silicon validation purposes. For instance, the idea presented in [157] is to use emulation to evaluate the critical-path timing coverage of a validation plan. Likewise, the authors in [158] use emulation for selection of coverage points with small overhead in order to assess the quality of test vectors by measuring metrics such as code coverage. However in this work and in particular in this chapter, we are not concerned about the quality of the test vectors nor we assess functional or code coverage. Hence first in section 4.2, we introduce our in-house assertion synthesis tool that is tuned towards the category of the assertions that are used in this work hence removing the need for using third-party tools. Afterwards, in section 4.3, we present an automated methodology to design emulation-ready hardware architectures that can be merged into the methodology that was introduced in chapter 3 to improve the accuracy of selecting assertions that are going to be instrumented as on-chip monitors. Following to this, we discuss our experimental observations in terms of enhancements to both the run-time and coverage metrics in section 4.4. Finally we close the chapter in section 4.5 by discussing the concluding remarks.

4.1 Background

The objective of the work presented in this chapter is to improve the accuracy of the selection and assessment of hardware assertion checkers for bit-flip detection using FPGA-based emulation architectures. Since the systematic methodology presented in chapter 3 provides the basis of this work, we quickly summarize different steps involved in the proposed methodology by only highlighting the key points. The methodology is shown in Figure 4.1.

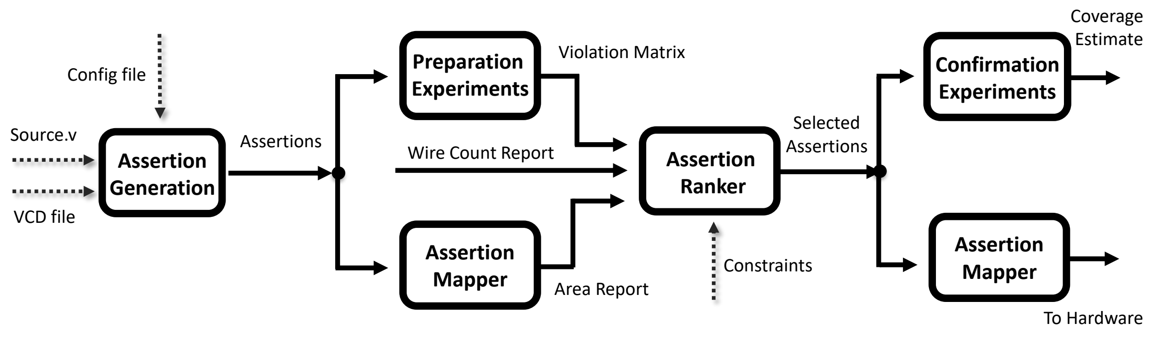


Figure 4.1: Tool flow for selecting the most suitable assertions to embed on-chip under wire constraints

1. **Assertion Generation:** Assertions are statements about the design's intended behavior that have to hold true indefinitely [27]. Traditionally, register-transfer level (RTL) assertions have been prepared manually by designers to improve observability and reduce debug time. Though, in recent years, techniques have been introduced for automatic assertion generation [96, 3, 2] with the goal of discovering non-obvious assertions in an automated way. This, for example, minimizes the amount of work for re-verifying a sub-block of the design after a revision. Since post-silicon validation is carried out after tape-out (netlist is ready), it is logical to configure the assertions generation tool to generate assertions that are based on the netlist rather than the behavioral model of the design.

2. **Preparation Experiments:** The very purpose of using high-level assertions in post-silicon validation is to convert them to their equivalent hardware circuit (as discussed later), for detecting bit-flips during post-silicon validation. Since assertions are properties that have to hold indefinitely, bit-flips that change the logic relationship between signals that construct an assertion will likely cause

the respective assertion to fire. Since it is impractical to map all the discovered assertions to hardware due to limited area and wiring budgets, only a subset of these assertions should be selected through a ranking process. However, before one can grade assertions, their potential in detecting bit-flips must be determined. During preparation experiments, the RTL model of the design is instrumented with all the assertions that have been discovered in the previous step. Afterwards, K (configured by the user) number of bit-flips are uniformly injected at random times in all the flip-flops of the design. The design is then simulated using random stimuli during which the status of all the embedded assertions are monitored. Once all the errors are injected, the relationship between assertions and flip-flops are formulated in an $M \times N$ matrix, where M is the total number of flip-flops and N is the total number of assertions. This matrix is called the *Violation Matrix*. For example, for a hypothetical circuit, in Figure 4.2, the second element in the first row reflects that, for all injections in flip-flop 1, assertion 2 has been violated twice whereas there is no violation in other assertions. This means that assertion 2 has the potential for detecting bit-flips in flip-flop 1. The *accuracy* of the violation matrix depends on the number of errors that are injected; the more error injections, the higher the chance for all the assertions to be exercised. However, since the run-time of this step depends on the number of flip-flops and the number of assertions, as the number of error injections increases, the run-time would increase too. By relying only on functional simulators, the run-time can quickly become impractical. For example, our experiments on ISCAS s38584 benchmark circuit [8] with around 18,000 assertions, show that for only 5 error injections per flip-flop, the total run-time of preparation experiments exceeds 50 hours. Trying to address this limitation is the key motivation for the work presented in this chapter.

$$\begin{array}{c|ccccc}
 & \textit{assr 1} & \textit{assr 2} & \textit{assr 3} & \textit{assr 4} & \textit{assr 5} \\
\hline
\textit{flip-flop 1} & 0 & 2 & 0 & 0 & 0 \\
\textit{flip-flop 2} & 0 & 0 & 1 & 0 & 0 \\
\textit{flip-flop 3} & 0 & 0 & 0 & 0 & 0 \\
\textit{flip-flop 4} & 3 & 1 & 2 & 0 & 4
\end{array}$$

Figure 4.2: Violation Matrix for a circuit with 4 flip-flops and 5 assertions.

3. **Assertion Mapper:** Assertions used for pre-silicon verification are not normally part of the synthesizable portion of the design. RTL assertions are written in high-level formats, such as property specification language (PSL) [159] or System Verilog Assertions (SVA) [160]. These formats are not directly synthesizable to hardware. However, recent research [134, 161] has shown methods to synthesize assertions to their equivalent hardware that can be embedded to the design for on-line property checking. The outcome of the assertion mapper is also the area overhead estimate for each assertion that is subsequently used by the assertion ranker.

4. **Assertion Ranker:** The potential of all the assertions in detecting bit-flips in flip-flops is captured by the violation matrix. Since it is impractical to add all the discovered assertions to the design, the ranker will select a subset of these assertions under a wire budget constraint. The ranker can be configured to meet two different goals; *bit-flip coverage estimate* maximization or *flip-flop coverage estimate* maximization. The mentioned coverage metrics are explained

in section 3.1.6 of chapter 3. The detailed discussion of the different ranking algorithms is out of the scope of this chapter and details can be found in section 3.1.5 .

5. **Confirmation Experiments** : Confirmation experiments assess how often the injected bit-flips are caught by the subset of the assertions that are selected by the ranking algorithm. There are two objectives in running confirmation experiments; firstly, it is a way of assessing the effectiveness of the ranking algorithm and assuring that the selected assertions meet their intended expectations in detecting bit-flips. Secondly, unlike pre-silicon verification and manufacturing test that benefit from well-defined and universally-adopted coverage metrics, there is no such metric to assess the effectiveness and completeness of post-silicon validation. It is not clear how to answer questions such as: ”how comprehensive the validation sessions are?” or ”are we done yet?”. In quest of filling this gap, we introduced *bit-flip coverage estimate* and *flip-flop coverage estimate* in section 3.1.6. These metrics are related to the post-silicon validation's error space (bit-flips in flip-flops). *Bit-flip coverage estimate*, is defined as the ratio between the number of bit-flips detected by the selected assertions and the total number of bit-flips injected during the confirmation experiments. For example, for the circuit in Figure 4.3 with three flip-flops, if 4 errors are injected in each flip-flop (12 injections in total), the bit-flip coverage estimate would be 5/12 = 42%. Bit-flip coverage estimate is a useful metric that can provide internal feedback to different steps in the entire methodology (e.g. ranker in Figure 4.1). Nonetheless, since post-silicon validation sessions run for extensively long durations, if at least one bit-flip of all the bit-flips that occur in a flip-flop is caught

by an embedded assertion, this still represents critical information concerning subtle design problems. Therefore, *flip-flop coverage estimate* is defined as the ratio between the number of flip-flops for which at least one of the injected bit-flips in them has been detected (at least one assertion violation) over the total number of flip-flops. For instance, referring again to Figure 4.3, it can be seen that there is no bit-flip detection for the second flip-flop. So, the flip-flop coverage estimate is calculated as $2/3 = 66\%$. Note, since the purpose of post-silicon validation is not to contain or recover from every bit-flip but rather to find out if bit-flips occur and, if they occur, to collect critical information for root-cause analysis, we consider flip-flop coverage estimate as a more suitable metric than bit-flip coverage estimate. As a final note, the reason we use the word "estimate" in both coverage metrics is because of the fact that unlike stuck-at faults in manufacturing test, bit-flips are single-cycle transient errors and if a bit-flip in a flip-flop is detected by an assertion at a particular time, there is no guarantee that another bit-flip in the same flip-flop is detected by

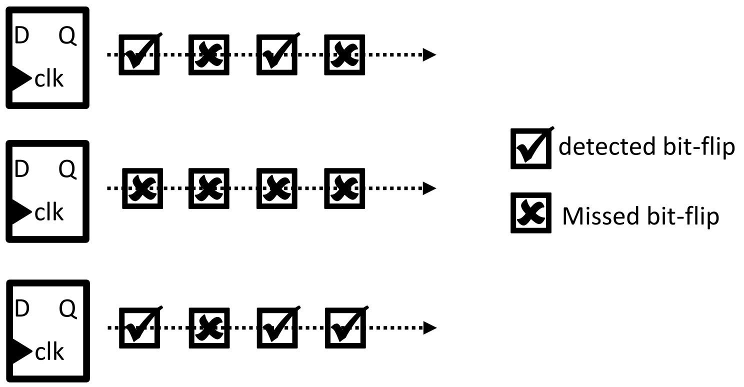


Figure 4.3: Hypothetical example for representing the difference between bit-flip and flip-flop coverage estimates.

that particular assertion. As it will be discussed later, this is due to the fact that an assertion can only detect a bit-flip if all the antecedent signals of the assertion are satisfied.

This section reviewed various steps that are carried out to select a subset of assertions, namely *Candidate Assertions* in Figure 4.1. The reader is referred to chapter 3 for the a more detailed discussion of our proposed methodology. In the following sections, we will elaborate on the new features that can be integrated to the different boxes in Figure 4.1 and lead to more accurate results in terms of coverage estimate, while also reducing the amount of on-chip area needed to accommodate the embedded hardware assertions.

4.2 Assertion Synthesis

As stated in section 4.1, RTL assertions are written in either PSL or SVA formats which cannot be directly synthesized to hardware. Although these high-level assertions monitor design's behavior and provide useful feedback in pre-silicon verification, in order to perform on-chip property checking during post-silicon validation and debug, their equivalent hardware must be generated and integrated to the circuit under validation (CUV). In this section, a simple assertion synthesis method which is tuned towards the assertions used in our work is introduced. It is important to note that our synthesis tool only supports a subset of assertions and by no means it should be regarded as a complete assertion synthesis tool. As it will be elaborated, in addition to generating the hardware equivalent of the assertions, we use our method for fast estimation of the area overhead of each individual assertion.

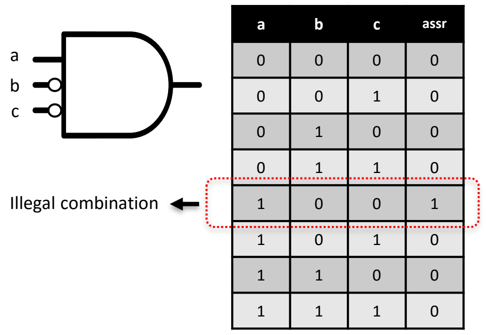
The problem of automating the generation of equivalent hardware units of pre-silicon assertions has recently been well studied (e.g., [134], [162], [163] and [135]). Most of these tools support sophisticated features such as sequences, repetitions, first occurrence matching, assertion covers and etc. However, since the assertions that are employed for bit-flip detection have a simplified structure (an antecedent condition implies a consequence), in this section we propose our own custom SVA assertion synthesis algorithm. It is important to note that the resulting circuit incorporates only those operators and constructs that are required by the assertions used for bit-flip detection and therefore additional features that serve purposes other than what is needed in this work are not taken into consideration. One objective of removing the unnecessary features is to reduce the area overhead. In addition to this, the other motivating factor for using an in-house assertion synthesis algorithm is that, we can estimate each assertion's area overhead because we know the exact number of flip-flops, inverters and the size of the AND gate that constructs the assertion. Since the ranking algorithm does relative area comparisons in between the assertions, it is sufficient to work out the estimated area overhead as long as the method is consistent for all the assertions. This will eliminate the need to pass the equivalent hardware unit of the assertions through commercial synthesis tools to achieve their area estimates (needed by the ranking algorithm), which, as was shown in section 3.3.3, is a time-consuming process specially with the large number of assertions in preparation experiments.

The focus of this section is on assertion synthesis and therefore, for the sake of completeness, we quickly review the structure of SVA assertions. This provides the necessary foundation for understanding the different steps in our assertion synthesis flow. Figure 4.4 represents a type of an SVA assertion that we typically encounter for bit-flip detection. In SVA, the ## construct is called the *cycle-delay* construct [27] and the number followed by ## represents the cycle in which the right-hand side Boolean event must occur with respect to the left-hand Boolean event. In addition, as shown in Figure 4.4, the signals are grouped based on whether they are at the left side of the implication operator (antecedent signals) or at the right side of it (consequent signals). SVA provides two implication operators |–> and |=>. The former is called the *overlapped implication* operator which means that if the left hand side prerequisite sequence holds, then the right hand side sequence must hold. The latter is called the *non-overlapped implication* operator which is similar to the overlapped operator except for the fact that the right-hand side sequence is evaluated in the next clock cycle. Hence a => b and (a |-> ##1 b) are equivalent.

$$\textbf{assert property (}\overbrace{\textbf{(a==1) \&\& (b==0) \#\#1 (c==1)}}^{\textbf{Antecedent Signals}}\ \textbf{|=>}\ \overbrace{\textbf{(d==}}^{\textbf{Consequent Signals}}\textbf{ 0)));}$$

Figure 4.4: Example of an System Verilog Assertion (SVA assertion).

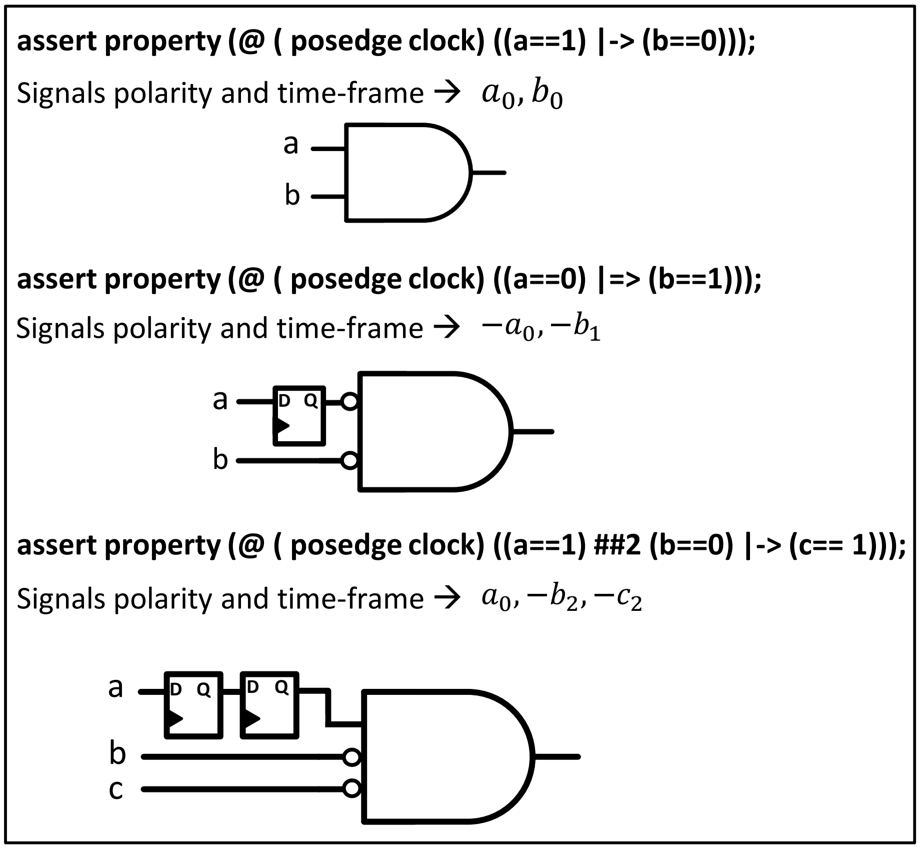
Now let us assume for the sake of simplicity that we want to generate the hardware circuit for an assertion with all its signals in the same clock cycle (temporal depth of zero), *assert property* $((a == 1)\ \&\&\ (b == 0)\ |{-}{>} (c == 1))$. This assertion is read as: if signal a is 1 and signal b is 0, signal c is implied to be 1. Now since we know that so long as a is 1 and b is 0, signal c is 1, a hardware circuit that evaluates to logic 1 when the assertion is violated can be constructed using a 3-input AND gate as shown in Figure 4.5. The output of the assertion should go to 1 when at the same clock cycle, signals a and b are 1 and 0 respectively but signal c is 0 instead of 1. Hence, in order to trigger that the assertion is violated, the complement of signal c must be connected to the AND gate. The output of the hardware circuit associated with the assertion will always be 0 as long as a, b and c contain legal values. As shown in Figure 4.5, only one entry in the associated truth table of the circuit evaluates the output to 1 and that entry is known as the illegal signal combination.



a	b	c	assr
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Figure 4.5: Example showing the equivalent hardware circuit for the SVA assertion "*assert property* $((a == 1)\ \&\&\ (b == 0)\ |\text{->}\ (c == 1))$. "

For all the assertions that are to be synthesized, one has to find whether a signal is at the antecedent side or the consequent side, its polarity which determines whether it must be inverted or not when it is connected to the AND gate and lastly its time-frame. For instance, for the assertion that was just discussed, the required synthesis information can be encoded in an statement as $(a_0, -b_0, -c_0)$. The sign before a signal determines if the signal must be inverted or not when connected to the final AND gate. Likewise, the subscript represents the signal's time-frame. For assertions with non-zero temporal depth (assertions that span across multiple clock-cycles), signals with the highest time-frame are directly connected to the AND gate whereas other signals are buffered accordingly. Figure 4.6 provides three examples for generating hardware circuits for assertions with and without temporal depth of 0. As it can be seen, for each signal, the time-frame and the polarity is found and is subsequently connected



```
assert property (@ ( posedge clock) ((a==1) |-> (b==0)));
```

Signals polarity and time-frame → a_0, b_0

```
assert property (@ ( posedge clock) ((a==0) |=> (b==1)));
```

Signals polarity and time-frame → $-a_0, -b_1$

```
assert property (@ ( posedge clock) ((a==1) ##2 (b==0) |-> (c== 1)));
```

Signals polarity and time-frame → $a_0, -b_2, -c_2$

Figure 4.6: Example showing the equivalent hardware circuit for assertions used in this work.

to the AND gate. The algorithm for finding the required synthesis information for an assertion has several steps which are elaborated below.

1. For each signal s_i in the assertion statement (both antecedent and consequent sides), find it's time-frame and prepare a time-frame list $T_S = \{t_{s_1}, t_{s_2}, \ldots, t_{s_n}\}$ in such way that the time-frame for the leftmost signal is 0. While moving from the first antecedent signal to the last consequent signal, each time a cycle-delay construct is seen, the time-frame is increased by the value that follows the cycle-delay construct. Note that $|\!\!\Rightarrow$ is equivalent to ##1 $|\!\!-\!\!>$. For example, in the

third example of Figure 4.6, the time-frame of a is 0 while the time-frame of b is 2. The time-frame of the consequent signal c is also 2.

2. Prepare the list of signals in the consequent side $C = \{c_1, c_2, \ldots, c_n\}$ and antecedent side $A = \{a_1, a_2, \ldots, c_n\}$.

3. For each signal s_i whose time-frame is smaller than the maximum time-frame, create a shift register with the size equals to $(max(T_S) - t_{s_i})$. Signals with the maximum time-frame are connected directly to the AND gate with the correct polarity which is found in the next two steps.

4. Find polarity of signals in the consequent side $P_C = \{p_{c_i} \mid \forall c_i \in C \text{ if } (c_i == 1),\ p_{c_i} \leftarrow \texttt{false},\ \text{else}\, p_{c_i} \leftarrow \texttt{true}\}$. If polarity is `false`, the signal must be complemented and if it is `true`, it is connected as is. For instance, for the third assertion in Figure 4.6, the consequent signal c must be a 1, therefore its polarity is negative thus, its complement is connected to the AND gate.

5. Find polarity of signals in the antecedent side $P_A = \{p_{a_i} \mid \forall a_i \in A \text{ if } (a_i == 1),\ p_{a_i} \leftarrow \texttt{true},\ \text{else}\, p_{a_i} \leftarrow \texttt{false}\}$. For instance, for the third assertion in Figure 4.6, the antecedent signal b must be a 0, therefore its complement is connected to the AND gate. Note, whenever the output of the AND gate goes to 1, it means that the assertion has been violated.

The area estimate for each assertion is estimated based on:

$$Assr(i)_{area} = (\alpha \times F) + (\beta \times In) + (\gamma \times Inv) \tag{4.1}$$

where F represents the total number of flip-flops, Inv is the total number of inverters and In represents the number of inputs to the AND gate. α, β and γ are technology dependent coefficients that reflect the relative differences in the area of flip-flops, inputs to an AND gate and inverters. These three coefficients can be customized by the user based on the specific standard cell library that is employed.

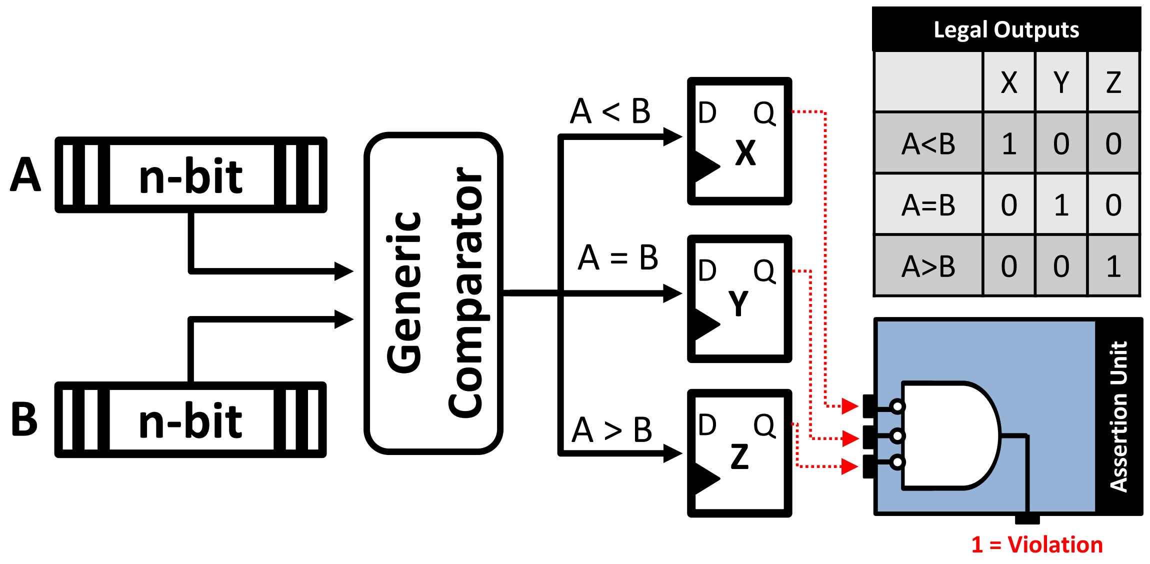
Finally, it is important to note that the very purpose of finding the area estimates is to provide necessary information for the cost function of the heuristic ranking algorithm detailed in section 3.1.5. It by no means provides the exact area overhead of the assertions, nonetheless rapid computation of estimates still captures the size of assertions relative to each other and therefore it avoids the need to use third party commercial synthesis tools to compute the exact area.

4.3 Architecture

In this section we present our proposed emulation-based methodology. At first, we highlight the benefits of emulation and subsequently detail the hardware architecture and the associated tool flow.

4.3.1 Benefits of Emulation for Assertion Assessment

Not only the functional simulators are slow for logic simulation, by adding a considerable number of assertions with the resulting overhead caused by concurrent property checking, the computational speed degrades significantly to the point that running experiments with more than 10 error injections per flip-flop, especially in preparation experiments (due to having a large number of assertions), becomes impractical. Unlike assertions in pre-silicon verification that are high-level behavioral property checking statements, the assertion checkers in hardware are logic blocks themselves. For instance, the digital circuit in Figure 4.7 represents a generic comparator that takes two unsigned n-bit registers as the input and depending on their value, provides a 3-bit output which is registered in one of the flip-flops marked as X,Y and Z. At any time, these flip-flops will store a one-hot code, as shown at the right side of the figure. One possible assertion for these outputs is to detect the all-zero state, which can be done in hardware with a 3-input AND gate with inverted inputs. In case there is a bit-flip in any of the flip-flops that is supposed to hold a 1, the output of this AND gate will be asserted. A critical benefit of assessing assertions in an emulation environment is that all the blocks run *concurrently*, hence there is no overhead in terms of the clock cycle count. Also, so long as the critical paths are in the circuit-under-validation (CUV) and not in the assertion blocks, there will be no penalty in terms of the clock frequency.



Legal Outputs	X	Y	Z
A<B	1	0	0
A=B	0	1	0
A>B	0	0	1

Figure 4.7: Example of instrumenting CUV with hardware assertions. In case there is a bit-flip in any of the X,Y or Z registers, the output of the assertion unit will evaluate to 1 which indicates a property violation.

Another benefit of emulation is to improve the density of the violation matrix, as illustrated in Figure 4.8. We have observed that the small number of error injections, which is limited in simulation-based experiments by their slow speed, affects the accuracy of the violation matrix. By increasing the number of error injections, the non-zero elements in the violation matrix will increase which clearly provide a more meaningful input to the assertion ranker. For instance, Figure 4.8 represents an emulation-based violation matrix, similar to the matrix in Figure 4.2, but with some elements having a larger value because of the increased number of injections that can be performed in emulation. Since even one error detection in a flip-flop is sufficient for considering that flip-flop as covered, the flip-flop coverage estimate will increase if the number of all-zero rows in this matrix decreases. Since the ranker select a subset of assertions for maximizing both the bit-flip coverage estimate and flip-flop coverage estimate, its accuracy is central the accuracy of the selected assertions.

$$\begin{array}{r c} & \begin{array}{ccccc} \textit{assr 1} & \textit{assr 2} & \textit{assr 3} & \textit{assr 4} & \textit{assr 5} \end{array} \\ \begin{array}{r} \textit{flip-flop 1} \\ \textit{flip-flop 2} \\ \textit{flip-flop 3} \\ \textit{flip-flop 4} \end{array} & \begin{pmatrix} 0 & 3 & 1 & 0 & 1 \\ 1 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 3 & 1 & 2 & 0 & 4 \end{pmatrix} \end{array}$$

Figure 4.8: An example of a violation matrix prepared by emulation-based experiments. The red elements are different from the elements in the same position in the matrix from Figure 4.2.

4.3.2 Hardware Architecture and Tool Flow

The direct benefit of running emulation-based preparation and confirmation experiments is the ability to inject a large number of errors in a considerably shorter time when compared to simulation-based experiments. This increases the likelihood of the circuit being pushed to the states that could not be reached in simulation due to limited number of applied vectors. Emulation has been extensively used in test for fault grading, e.g., [164], and fault-tolerant computing for assessing the impact of single-event upsets, e.g., [165]. Nonetheless, as articulated in the previous sections and chapters, post-silicon validation is a different problem and our proposed architectures and tool flow have been designed toward a better assessment and selection of hardware assertion checkers for bit-flip detection. Figure 4.9 overviews the architecture and the details for each block are elaborated below. As shown later, the internals of the sub-blocks in Figure 4.9 are different for preparation experiments where the focus is on grading all the assertions and confirmation experiments where the focus is on measuring the coverage metrics. In addition, the confirmation experiments themselves have different architectures for bit-flip coverage estimate and flip-flop coverage estimate.

Phase Locked Loop

For reasons that will be clarified in section 4.3.2, one of the proposed architectures operates using two clock domains. We have decided to use phase-locked loops (PLLs) mainly because of the following reasons: first, using divide-by-n clock division is prone to timing closure problems and secondly, the logic resources that are required to implement divide-by-n clock division can be saved by utilizing embedded PLLs that

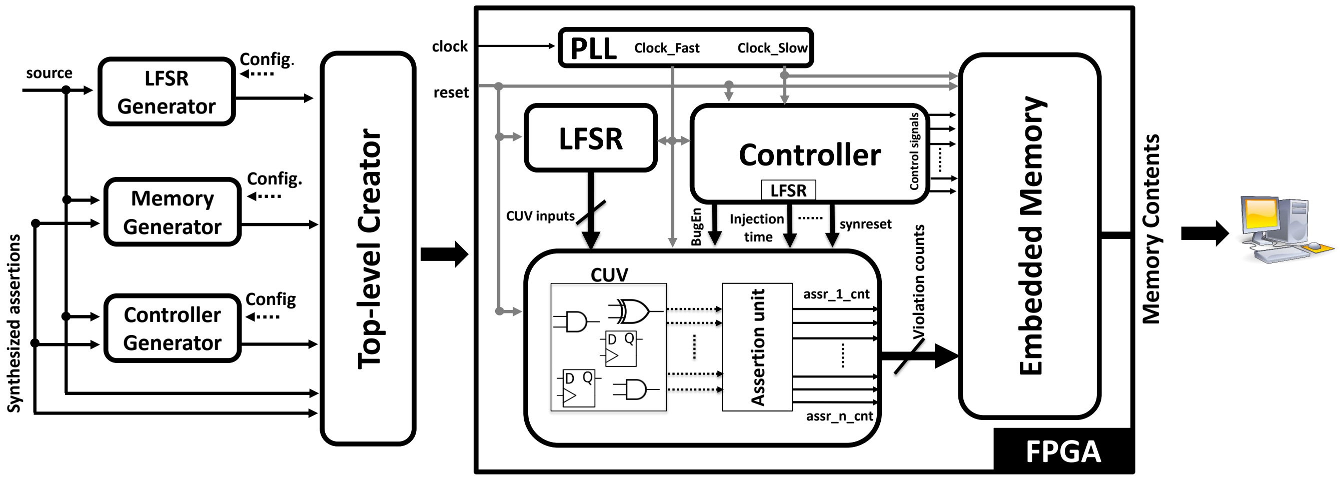


Figure 4.9: Tool flow for automatic generation of emulation-ready hardware architecture to accelerate error injection experiments in post-silicon validation.

are pre-fabricated on most FPGAs. Our current implementation will instantiate a PLL that generates a fast and a slow clock signal. The ratio between the fast and slow clocks are configurable by the user (dependent on the specific FPGA device that is used). Note that the side-effects of using PLLs such as increased energy consumption and routing difficulties for very large designs are out of the scope of this work.

Input Stimuli Generation

Since the input stimuli used in both preparation experiments and confirmation experiments are random, a PI-bit linear feedback shift register (LFSR) is created by our tool where PI is the total number of primary inputs in the design. The characteristic polynomials for LFSRs are primitive and irreducible in order to support the maximal-length sequences.

Circuit-Under-Validation and Assertion Unit

Prior to connecting the CUV to the assertion unit, each flip-flop in the design is instrumented with a 2-to-1 MUX to facilitate error injections. The inputs of the MUX are connected to Q and $\bar{Q}$ outputs of the flip-flop. The select signal of MUX, which is asserted through the controller (detailed below) based on the logic values of *BugEn*, *injection time* and *synreset* signals, determines when a bit-flip should occur. Note that all bit-flips that are injected are single-clock-cycle bit-flips. In addition, the assertion unit which is the synthesized hardware of the SVA assertions is connected to the CUV as shown in Figure 4.9.

Controller

The objective of the controller is to determine when and where a bit-flip should occur, monitor the status of assertions after bit-flip injections and initiate a burst of writes to the memory. The injection time is determined by a 20-bit LFSR, inside the controller whose initial state is configured randomly by our proposed tool prior to compilation. Once the entire platform starts running, the controller will inject E number of errors in the first flip-flop at different times which have been determined by the LFSR within the control unit. E is selected by the user and it is passed as an option to the proposed toolflow. After each error injection (a single cycle bit-flip), the circuit continues to run for a predefined number of clock cycles (given by the user) during which the controller activates the assertion unit to monitor if there are any violations. Due to the evident differences in the objectives of preparation experiments and confirmation experiments, the sequence of writes to the memory are done through two different architectures.

(a) **Preparation Experiments architecture:** For each assertion there is a counter which determines the number of times the respective assertion has been violated. Once all the errors are injected in a flip-flop, the controller stops the circuit's operation. At this time, the controller will check the violation count register of each assertion and organizes memory writes in case a non-zero value is observed. The controller's interface to the memory can be configured in two different ways. The first configuration utilizes two clock-domains in such way that the sequence of checks for each assertion is done through a second finite-state machine (FSM) which works based on a slower clock as shown in Figure 4.11. The main reason for using a slower clock for this part of the design can be explained as follows. For the sake of argument, assume that we inject E number of bit-flips in one of the flip-flops in a design and that the design has M number of assertions. After all the injections have been done in a flip-flop, the controller will have to check the violation counter of every single assertion. Since a fairly large number of assertions are instrumented to the CUV, which need to be checked one by one, and also the fact that there is a single embedded memory that can accommodate one write at a time, the multiplexer that determines the words that are to be written to the memory can become reasonably large as shown in Figure 4.10(a). This can result in a timing closure problem due to long propagation delay in this multiplexer. One has to note that the number of clock cycles spent on checking the status of each assertion counter is bounded by the number of assertions (in this case M), which is significantly less than the number of clock cycles needed for error injections ($E\times$ number of clock cycles between error injections). Therefore it is sensible to keep the clock frequency of error injection block (CUV and assertion checker unit) as high as possible and adjust the clock frequency of the assertion checking FSM as shown in Figure 4.11.

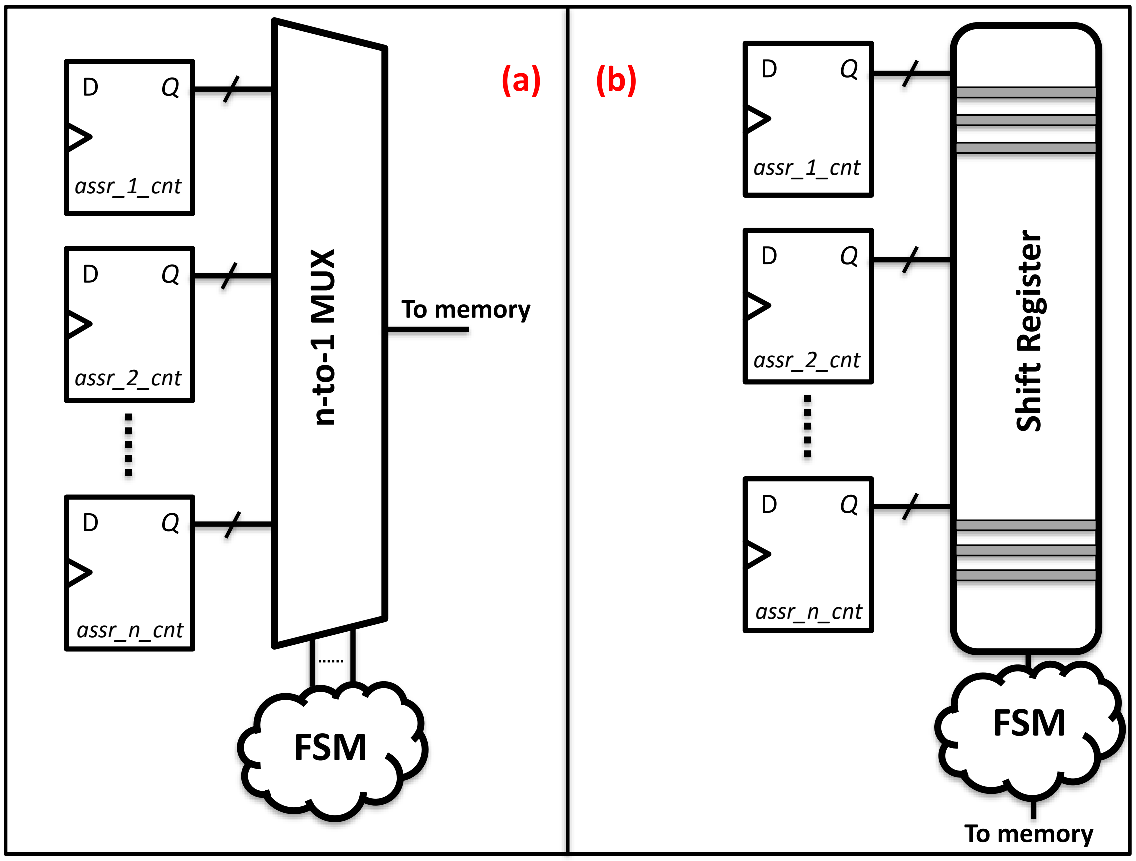


Figure 4.10: Memory interface for two-clock domain (a) and single-clock domain (b) architectures in preparation experiments. Note that flop ID and assertion ID are concatenated to the output of the MUX which then will be written to the memory.

The following example shows that using a slow clock to write the assertion counts to memory will have a negligible impact on the overall emulation time. Assume that we want to run emulation experiments for a circuit with 1000 flip-flops (F) and 400 assertions (A) and that 1024 error injections (E) are scheduled to be injected in each flip-flop. The insertion time is generated by a

20-bit LFSR. Let us assume that on average, 2^{10} clock cycles (CC) are elapsed for each error injection. Hence the total time spent for error injections based on a 50 MHz clock (F_{50}) is:

$$T = F{\times}E{\times}CC{\times}\frac{1}{F_{50}} = 10^3{\times}2^{10}{\times}2^{10}{\times}20\,ns = 20.9\,s$$

Now the number of clock cycles that are spent for checking assertions after error injections in a single flip-flop is based on the total number of assertions. Hence, the amount of time spent in this step for error injection in all flip-flops (F) for a 50 MHz and a 1 MHz clocks are:

$$T^{Assr}_{clock50} = F{\times}A{\times}\frac{1}{f_{50}} = 1000{\times}400{\times}\frac{1}{50 \times 10^6} = 8\,ms$$
$$T^{Assr}_{clock10} = F{\times}A{\times}\frac{1}{f_1} = 1000{\times}400{\times}\frac{1}{01 \times 10^6} = 400\,ms$$

Based on the observations in the example above, it can be concluded that the performance penalty of using a slower clock for memory writes (in this case 1 MHz clock) is only 1.8%. On the other hand, if the clock frequency is slowed down to accommodate the slow paths in the memory writing FSM, the performance penalty will scale linearly with the increase in the clock period. For a circuit of a size like the one in this example, we have observed that the clock period can become three times longer.

It is worth mentioning that so long as the ratio between the two clocks is a natural number, synchronizers will not be needed between the two clock domains.

This is because the handshaking signals (dotted arrows in Figure 4.11) that facilitate the communication between the faster clock FSM and the slower clock FSM are not acknowledged until they are captured by the slower clock FSM and the other way round. One might argue that the slower clock can be avoided by pipelining this large multiplexer in such way that one write is performed in each clock cycle. Although this will lead to faster memory updates, using this approach will result in significant on-chip area usage associated with pipelining registers and logic (recall that the number of inputs to the multiplexer is in the range of hundreds). Therefore we choose to operate this FSM at a lower frequency (thus avoiding the registers needed for pipelining).

For applications where only one clock domain is available, the architecture can be configured in such way that, once all the errors are injected in a flip-flop, the contents of each assertion's violation counter is stored in a shift-register structure as shown in Figure4.10(b). Afterwards, contents of this shift-register are offloaded to the memory one-by-one at every clock cycle. The small boost to the run-time is significantly outweighed by the resulting resource usage overhead imposed by the size of shift-register that has to account for the worst-case scenario (many violations per error injections). Therefore, the author prefers the fast/slow clock method, especially because the performance penalty is negligible. Finally, please note that, as mentioned earlier, the violation matrix that is created in preparation experiments provides the necessary information for the assertion ranker to select a subset of assertions that are used in confirmation experiments to measure the bit-flip coverage estimate and the flip-flop coverage estimate.

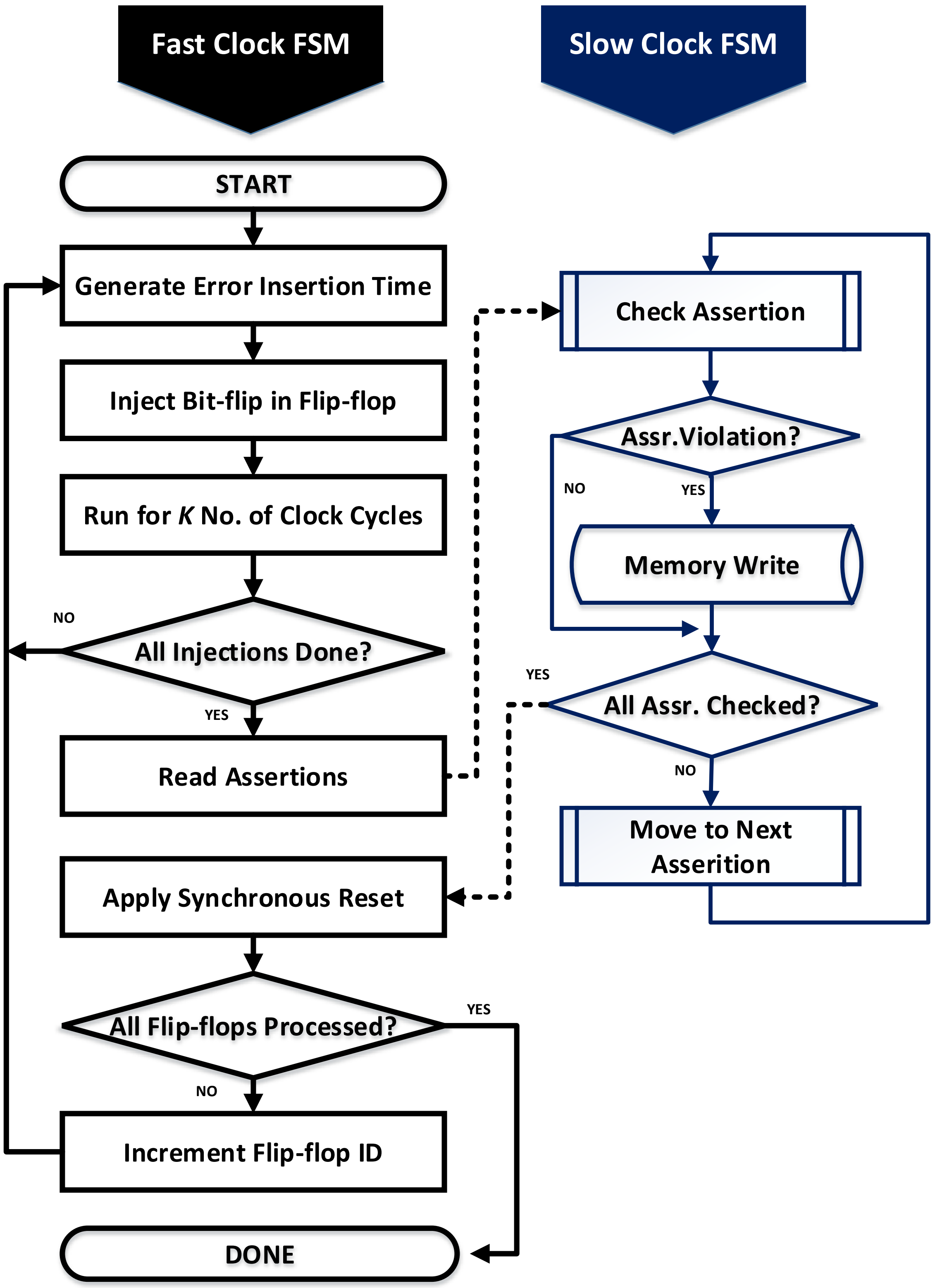


Figure 4.11: Controller FSM showing two different state machines for fast and slow clocks. Note that k is user defined and represents the number of clock cycles for which the circuit runs after each bit-flip injection.

(b) **Confirmation Experiments architecture:** As explained in section 5 and in detail in section 3.1.6 of chapter 3, the focus of the confirmation experiments is on the assessment of the assertions that have been selected by the ranking algorithm. The metrics *bit-flip coverage estimate* and *flip-flop coverage estimate* will determine whether the selected assertions fulfill the required expectations or not (these metrics were introduced in section 3.1.6 of chapter 3). For computing the flip-flop coverage estimate the number of times these assertions have been violated is of no importance, unlike preparation experiments where for the sake of grading, the number of times an assertion has fired will increase its bit-flip detection potential and hence is important. As a result, the violation counters of the assertions are removed and the assertion outputs are connected to a flag register that becomes 1 (and stays 1) if at least one out of all the bit-flips in the target flip-flop violates that assertion. A memory write is organized if the output of the OR gate in Figure 4.12(b) is 1. The contents of the registers are reset when all the bit-flips are injected in a flip-flop, the longest sequential depth of all the assertions has passed and the memory write has been performed. For bit-flip coverage estimate, on the other hand, the status of assertions must be checked every time a bit-flip occurs in a flip-flop. Hence, after each bit-flip, a memory write is organized if the output of the OR gate in Figure 4.12(a) is 1. The contents of the registers are reset every time a bit-flip is injected and the longest sequential depth of all the assertions has passed. Note, eliminating violation counters will be particularly useful for minimizing the on-chip memory usage overhead as well as compilation time for generating the FPGA bitstream.

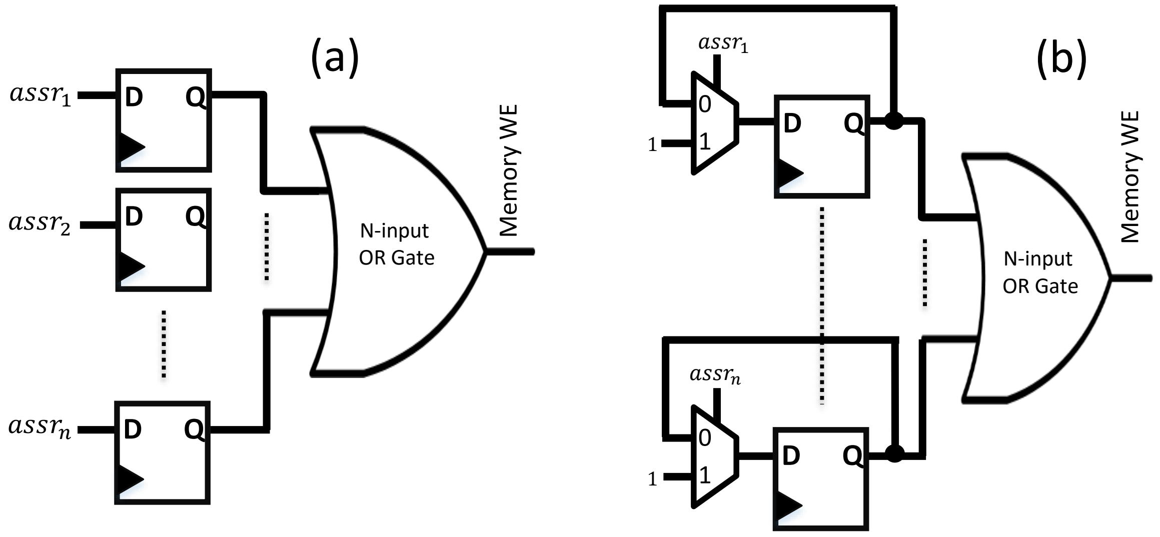


Figure 4.12: Memory interface for (a) bit-flip coverage estimate and (b) flip-flop coverage estimate. When write enable is granted, a single-bit 1 is written to the memory.

Memory Unit

During both the preparation and the confirmation experiments, the information related to the bit-flips that have been detected by assertions are stored in the embedded memory. The largest amount of information that needs to be stored is in preparation experiments. This is because, in order to assess the quality of assertions, one has to know the flip-flops they can potentially cover (detect bit-flips in that flip-flop) and the number of times they can do it when multiple errors occur in the same flip-flop. Hence, as shown in Figure 4.13(a), each word in the memory contains information about the flip-flop, the assertion ID that has caught the bit-flips in that flip-flop and the number of times that assertion has been violated. The width of Flop ID and Assr. ID are configured based on the total number of flip-flops in the design and the total number of assertions that are embedded in hardware. The width of Violation Cnt.

segment is determined by our tool based on the maximum number of error injections. In order to determine the memory depth, we used our findings from simulation-based experiments. Based on those observations, the expected number of assertions that would fire for each bit-flip injection is set to 10. Therefore, we used this coefficient for accommodating enough space in the available physical memory. Though, for debug sessions that have a large number of assertions and using 10 as the coefficient would result in the memory not fitting on the target FPGA device, smaller coefficients in the range of 1 to 10 is used. The tool flow has an automatic mechanism such that if memory overflow occurs before all the flip-flops are evaluated, the session is divided into multiple sessions, with less assertions in each session. Clearly, this coefficient and the number of assertions that can be mapped to the device in a single session depends on the capacity of the target device.

During confirmation experiments, only the information needed to determine bit-flip coverage estimate and the flip-flop coverage estimate are stored. For bit-flip coverage estimate, every time a bit-flip is injected in a flip-flop and the largest sequential depth of all assertions is passed, the output of the OR gate in figure 4.12(a) is evaluated such that if it is 1, it implies that the bit-flip has been detected and a single-bit 1 is written to the memory in a sequential manner. In an ideal case where all bit-flips are detected, the maximum required memory depth equals the total number of flip-flops multiplied by the number of errors that are injected in each flip-flop. For instance, for a design with 20 flip-flops, if 5 bit-flips are set to be injected, the required memory size would be 100 bits.

On the other hand, for flip-flop coverage estimate, after all the bit-flips are injected in a flip-flop, if the output of the OR gate in Figure 4.12(b) is 1, then that flip-flop is

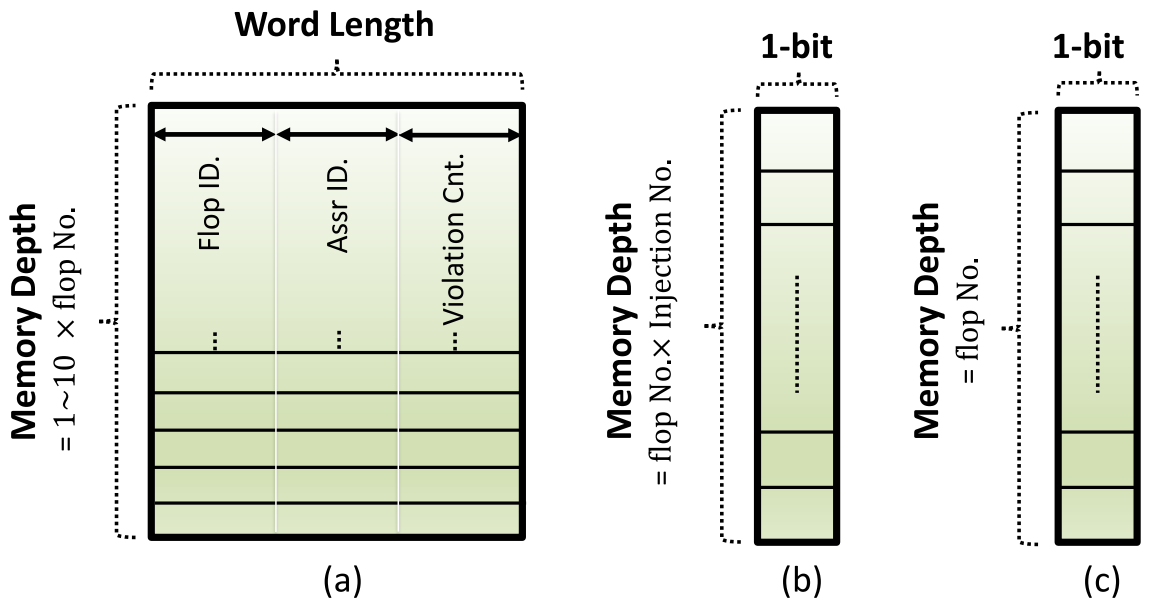


Figure 4.13: Memory layout for (a) preparation experiments, (b) bit-flip coverage estimate in confirmation experiments and (c) flip-flop coverage estimate in confirmation experiments.

marked as potentially covered. Since we want to determine how many flip-flops are potentially covered, it would be sufficient to store a single-bit 1 every time a flip-flop is marked as covered. Therefore, the maximum required memory depth is the total number of flip-flops. Clearly, the memory address at which a 1 is written corresponds to the ID of the covered flip-flop. Memory layouts for confirmation experiments are shown in Figures 4.13(b) and 4.13(c).

4.4 Results and Discussion

The proposed tool flow has been implemented on an Intel core i7 machine with 32GB of RAM using GCC 4.8.4 for compiling C++ source codes and Tcl 8.5 and Python 2.7.6 for scripting. For assertion discovery, we have used GoldMine [3], an automatic assertion generation tool which has multiple built-in data mining engines for finding the likely design invariants. These likely invariants are passed through a formal verification tool that filters out the incorrect invariants leaving true invariants as design's final assertions. Since GoldMine operates based on simulation traces, results from both the random input (Goldmine's default) stimuli and the deterministic vectors produced by Validation Vector Generator tool from Virginia Tech [150] have been provided to GoldMine's mining engine. Assertions from these two methods have been merged as are the assertions from different mining engines of GoldMine. Details about the specifics of GoldMine are out of the scope of this work and the interested reader is refered to [3] and [62].

In order to generate the equivalent hardware circuits of the high-level SystemVerilog [166] Assertions (SVA assertions), we have implemented the algorithm that was detailed in section 4.2 in such way that, initially, all the assertions that have been found using GoldMine are added to the source code and passed to our tool to produce their equivalent hardware description language (HDL) description and their area estimate (required by the ranking algorithm). Once a subset of these assertions are selected through the ranking algorithm, namely candidate assertions in Figure 4.1, we will instrument them to the original design and pass them through Synopsys Design Compiler to find the accurate area overhead, which takes into account the logic sharing in between the assertions. This is a more realistic measure of the area overhead (due to logic sharing between assertions) than the trivial method of summing up the area overheads of the individual assertions.

To support the random occurrence of bit-flips in post-silicon validation, we have used random input stimuli throughout both the preparation and confirmation experiments. As explained in section 4.3.2, the LFSR unit is designed to produce random vectors whose bit-width is equal to the number of CUV inputs.

We have deployed our architecture on an Altera DE2 Cyclone IV device [167] with a 50 MHz reference clock. All memory dumps have been done using the in-system memory content editor feature of Quartus that operates through the JTAG port. Finally, the faster clock in our controller in preparation experiments is a 50 MHz clock and the slower one is a 1 MHz clock (both coming from a PLL unit), and the confirmation experiments operate on a single 50 Mhz clock.

The most important outcome of emulation (and the key motivation for its usage) is the ability to create a more accurate violation matrix after preparation experiments. This, in turn, results in an improvement in the selection of assertions that are to be mapped to hardware, which eventually leads to more precise coverage estimates. Therefore, we have integrated our proposed tool flow and architecture to the methodology shown in Figure 4.1 and ran preparation experiments with 256 error injections (discussed later) per flip-flop. Afterwards, we have passed the resulting violation matrix to the assertion ranker, which selects a different number of assertions by varying the wire count constraint. Following to this, we ran confirmation experiments on each set of these selected assertions and varied the number of error injections from 5 to 5000, something that is infeasible to do in simulation-based experiments. Table 4.1 provides the total number of assertions in preparation and confirmation experiments for the three largest ISCAS circuits. Please note that the number of assertions for preparation experiments is not bounded by the wire usage and is dependent on the assertion discovery engine whereas for confirmation experiments, the number of selected assertions is dependent on the ranker which takes into account the wire usage and the target coverage metric. In the following subsections, we will provide and discuss our experimental findings.

Wire usage	**Circuit**	**Preparation Experiments**	**Confirmation Experiments**	
			No. of assertions for bit-flip coverage estimate	No. of assertions for flip-flop coverage estimate
10%	s35932	8192	158	156
	s38417	4488	107	103
	s38584	28365	210	174
20%	s35932	8192	245	240
	s38417	4488	269	214
	s38584	28365	503	335
30%	s35932	8192	331	329
	s38417	4488	397	333
	s38584	28365	817	478
40%	s35932	8192	418	415
	s38417	4488	510	456
	s38584	28365	1114	633

Table 4.1: Number of assertions in preparation experiments and confirmation experiments for different wire usage.

4.4.1 Preparation experiments

Due to the limited capacity of FPGAs, it is not feasible to instrument all the assertions discovered by GoldMine and map them to the FPGA for the preparation experiments (see Figure 4.1 from Section 4.1). Hence, assuming that only one FPGA board is available, depending on the number of assertions and the FPGA capacity, the preparation experiments are divided into multiple emulation sessions as seen in

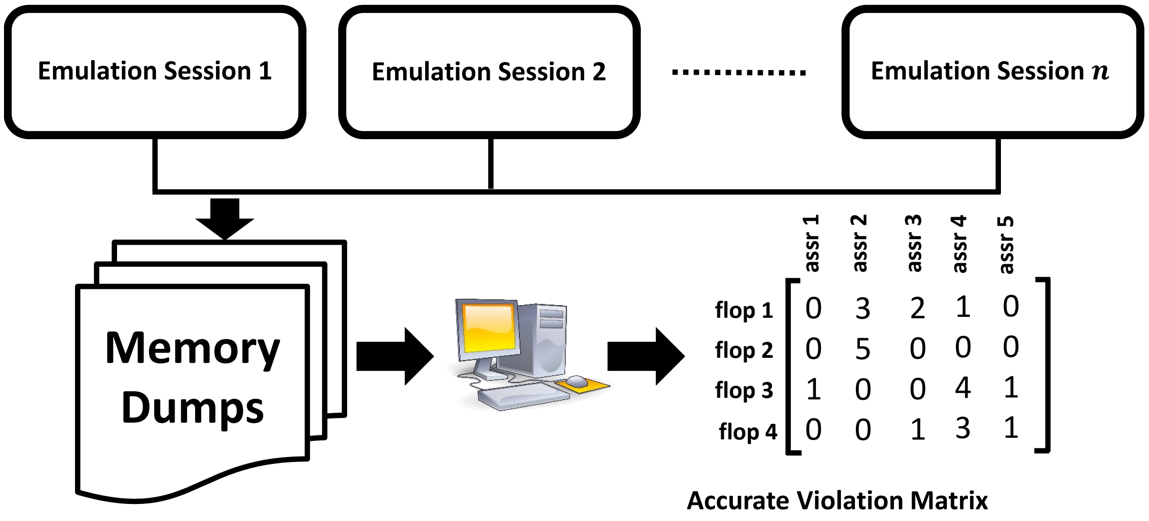


Figure 4.14: Running preparation experiments in multiple emulation sessions.

Figure 4.14. It is worth noting that, although one can have a single emulation session by using a high-capacity FPGA that can accommodate all the assertions that are mined, it is more practical to employ multiple FPGA boards (with devices of lower capacity) and run experiments in parallel because there is no dependency in between the different emulation sessions. Though in this work, we have used a single FPGA board and all our comparisons are based on measuring the total time spent in all the emulation sessions.

Table 4.2 shows the comparison between the total amount of time spent in simulation and emulation for running preparation experiments for the ISCAS89 s38584 benchmark circuit. The reason behind choosing this circuit for comparison is that it has the highest number of assertions compared to the other two ISCAS circuits that have more than 1000 flip-flops. Therefore, it will have the worst run-times both in emulation-based (larger number of emulation sessions) and simulation-based (larger number of assertions) experiments. Note that a similar trend also holds for s38417

Preparation Experiments			
No. of error injections	**Run-time**		**Flip-flop coverage estimate**
	Simulation (hours)	Emulation (hours)	
2	26.3	$45 \times 0.001 = 0.045$	79.6%
8	105	$45 \times 0.12 = 5.4$	86.15%
64	-	$45 \times 0.89 = 40.05$	88.56%
256	-	$45 \times 1.64 = 73.8$	88.94%

Table 4.2: Comparison of run-time and accuracy improvements for ISCAS s38584 circuit. The emulation (total) represents the total time spent on creating the top-level design, preparing the bitstream, compilation and on-board execution together with memory dumps.

and s35932 circuits from the ISCAS89 benchmark set [8].

As stated earlier, for the same number of error injections, the simulator run-times are longer for preparation experiments in comparison with the confirmation experiments. This is because of the fact that in preparation experiments, a large number of SVA assertions are added to the the design and the associated overhead caused by the concurrent property checking makes the simulators run much slower when a large number of assertions are added to the design. This is however not an issue in emulation because the assertions are synthesized to their hardware equivalent circuit and all the units (CUV and assertion units) are working concurrently at the same clock speed. Nevertheless, as shown in Figure 4.14, due to the limited FPGA device capacity and the large number of assertions that need to be evaluated in preparation experiments, the experiments are done in multiple sessions. For ISCAS s38584, the entire experiment was divided into 45 sessions. The reported run-time is the sum of time spent in each session. Due to long run-times, authors did not carry out simulation-based experiments beyond 20 error injections per flip-flop for preparation experiments. It is worth to note that even in cases with small number of error injections at which the FPGA compilation overhead might exceed the overall simulation time, if there exists a very large number of assertions, the simulator becomes too slow to the point that it will still be faster to run emulation experiments with the compilation overhead

In addition to the significant run-time improvements, it can be seen that the flip-flop coverage estimate also increases by 9%. This means that the ranker will benefit from a more accurate relation between flip-flops, assertions and their error space (Violation Matrix) which results in the selection of more accurate candidate assertions. Table 4.3 represents the maximum observed flip-flop coverage with 256 number of error injections in preparation experiments which indicates the experimental maximum possible flip-flop coverage in case all of the assertions are instrumented to the design. As it can be seen, ISCAS s38417 has the lowest maximum flip-flop coverage which is explained by the lower than usual (compared to other benchmark circuits) number of assertions that could be mined for this circuit.

Circuit	**Flip-flop Coverage (%)**
s35932	98.6%
s38417	54.7%
s38584	88.9%

Table 4.3: Maximum observed flip-flop coverage in preparation experiments for three largest ISCAS benchmark circuits.

4.4.2 Confirmation Experiments

During confirmation experiments, only the selected assertions from the ranker (candidate assertions in Figure 4.1) are instrumented to the design. Using our current setup

No. of Injections	**Wire Count (%)**	**Bit-flip Coverage Estimate (%)**		
		s35932	*s38417*	*s38584*
1	10	8.7%	2.2%	9.9%
	20	13.3%	8.4%	16.5%
	30	34.6%	13.2%	22.8%
	40	23.8%	23.1%	33.4%
2	10	13.4%	3.7%	9.7%
	20	20.4%	12.6%	20.3%
	30	26.1%	22.4%	25.1%
	40	33.8%	29.8%	31.4%
8	10	12.5%	3.5%	9.6%
	20	18.4%	12.4%	19.8%
	30	23.9%	20.8%	24.7%
	40	31.6%	28.7%	32.4%
256	10	13.1%	3.4%	8.4%
	20	19.8%	12.1%	19.1%
	30	25.3%	20.6%	23.4%
	40	30.2%	27.9%	30.9%

Table 4.4: Evaluation of bit-flip coverage estimate for different number of injections with different wire budget constraints for three largest ISCAS benchmark circuit.

and the selected FPGA device, there is no need to have multiple debug sessions as the entire architecture can be fit in a single debug session. Note that, as shown in Figure 4.13, the memory unit of confirmation experiments and its associated FSM are much simpler than those of preparation experiments. During confirmation experiments, the quality of the selected assertions are assessed using the two coverage metrics, bit-flip coverage estimate and flip-flop coverage estimate that were introduced in chapter 3 and discussed in summarized in section 4.1. First we evaluate bit-flip coverage estimate. The ranking algorithm in assertion ranker of Figure 4.1 is configured to select a subset of assertions aiming at maximizing the bit-flip coverage estimate. Results are shown in Table 4.4. The bit-flip coverage estimate is measured for 4 different wire budgets from 10% of total number of flip-flops to 40% when the number of error injections is increased from 1 bit-flip to 256 bit-flips per flip-flop. As it can be seen, in general, as the number of error injection increases, the bit-flip coverage decreases. This is because for a bit-flip to be detected, all the signals in the antecedent side of an assertion must hold their intended value. If at that particular time when a bit-flip is injected, the antecedent signals do not hold their supposed values, then the consequent signal is not evaluated, the assertion is not exercised and hence, the bit-flip will be missed. The associated area overhead for bit-flip coverage estimate for different wire budgets is shown in Figure 4.15.

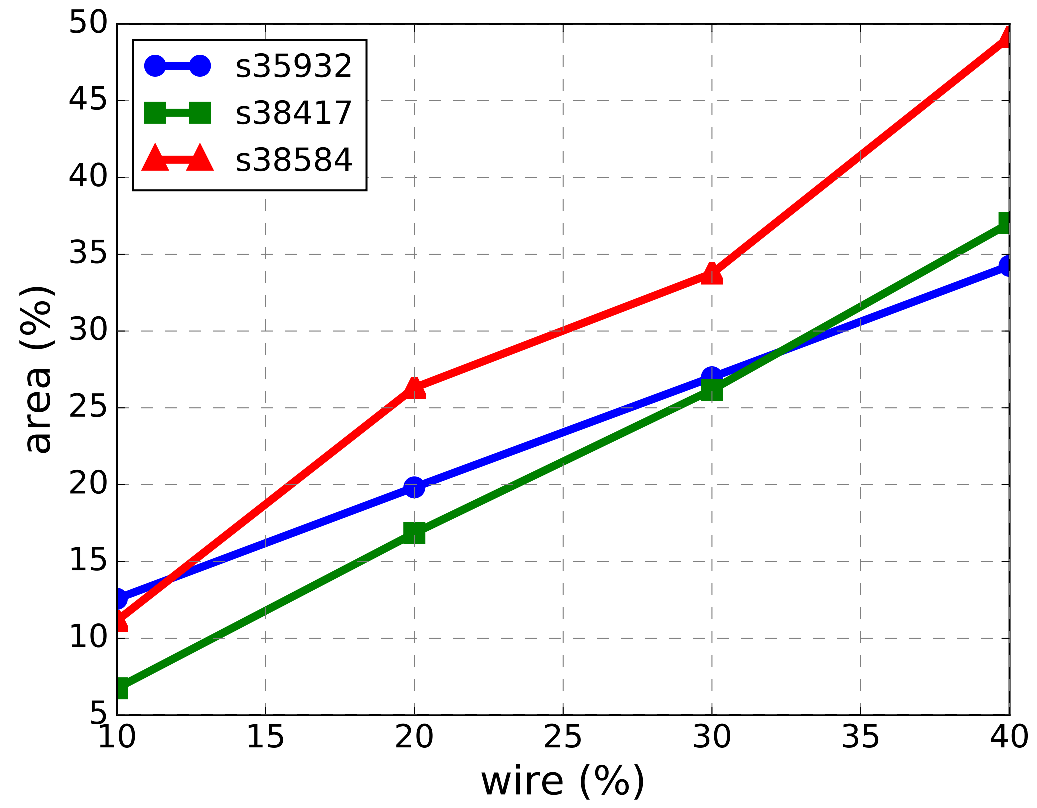
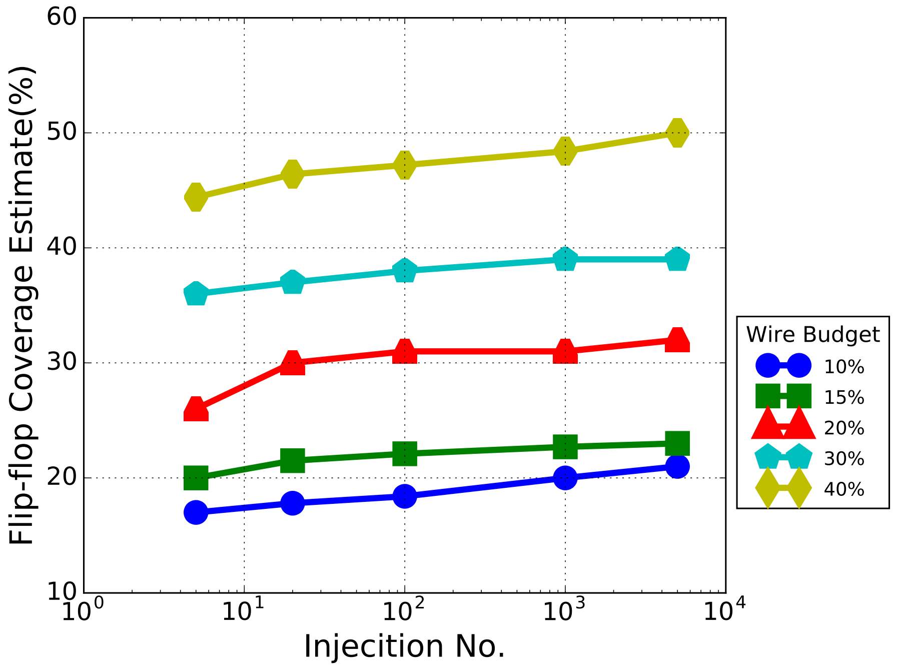


Figure 4.15: Evaluation of area overhead with respect to different wire budgets for the largest ISCAS circuits when the ranker is set to maximize bit-flip coverage.

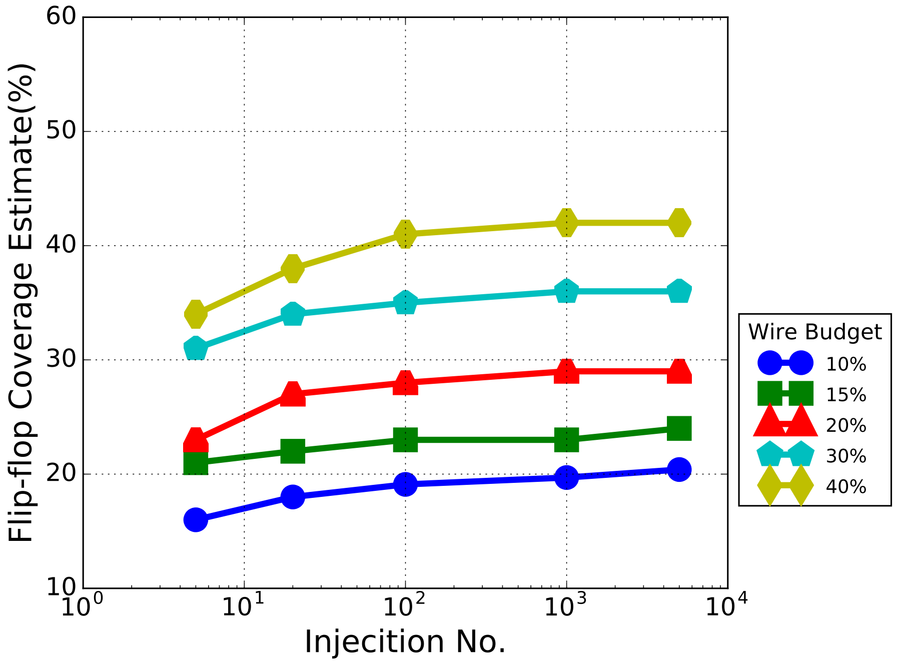
As explained in section 5, although bit-flip coverage estimate is an important metric that can be used to refine different steps in the entire methodology in Figure 4.1, flip-flop coverage estimate is a more relevant practical metric for the purpose of post-silicon validation. To justify this, one needs to review the key difference between the preparation experiments and the confirmation experiments. In confirmation experiments, if, out of the many errors that occur in a flip-flop, even one is detected by the embedded assertions, that flip-flop is considered as a potentially covered flip-flop. This is however not the case in bit-flip coverage estimate because the focus is on determining how many of the bit-flips are detected (covered) rather than flip-flops. Since post-silicon validation sessions run for extensively long durations (weeks or even months), even if one of the bit-flips in a flip-flop is detected by the assertions with a reasonably short detection latency, the embedded trace buffers will start recording relevant data that can later on be used for root-cause analysis [38, 142]. It is important to re-emphasize that the key objectives during post-silicon validation are to detect errors with low latency and record data for root-causing, which is significantly different from providing fault tolerant tolerant features, where correction of every bit-flip is important. Therefore, even if a small number of bit-flips are detected in a flip-flop over long validation sessions, the key objectives of post-silicon validation have been met.

Figure 4.16 shows the flip-flop coverage estimate for different wire budgets when the number of injections is increased from 5 error injections to 5000 error injections. As it can be seen, with small number of injections, there is a possibility that some assertions are never sensitized which leads to the inability to cover the flip-flops that are monitored by those assertions. There is a steep increase when changing the number of injections from 5 to 20, however the slope of the curves decreases as the number of error injections increases further. This is because for the majority of the flip-flops a bit-flip is detected in one of its first 20 occurrences in the respective flip-flops. Nonetheless, because bit-flip detection is dependent on the state of the circuit when the bit-flip occurs, for some flip-flops it might take a larger longer time until one of those corner states is reached that will cause the assertions to detect the bit-flip. Note, while the generation of long validation sequences for post-silicon validation is an important problem, it is beyond the scope of this study. We rely on extensive randomized validation sequences for our experiments and it is a normal expectation that focused sequences (that push the circuit in its corner states faster) will detect bit-flips even sooner.

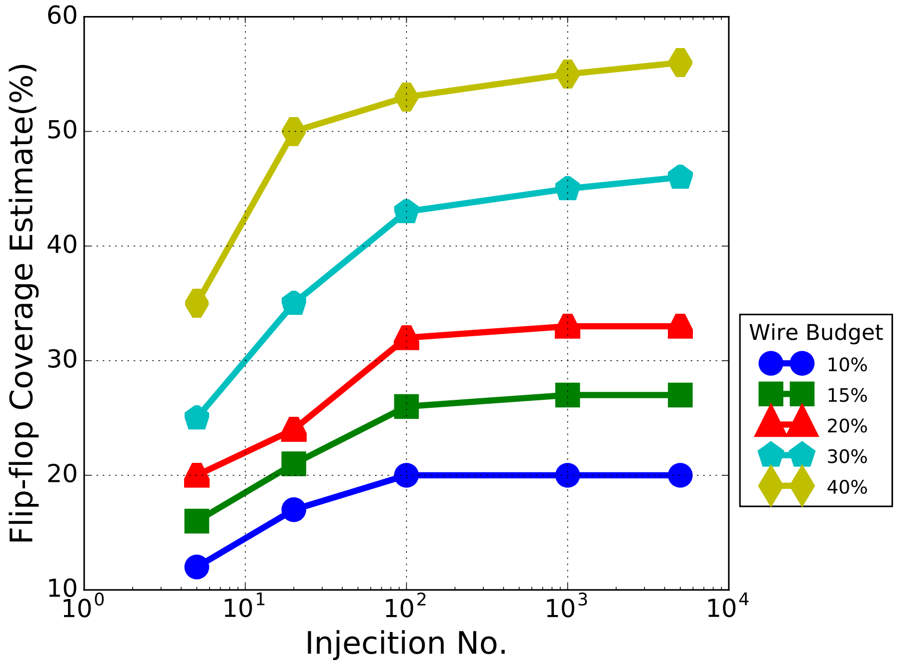
Finally, the area overhead for assertions that maximize flip-flop coverage estimate is shown in Figure 4.16d. It is worth noting that by using our custom synthesis tool instead of a generic tool such as MBAC [134], as used in chapter 3, the area overhead is improved on average by 4.2%. As explained in section 4.2, the main motivating factors for designing our custom synthesis tool are 1) to abandon the need to pass all the discovered assertions through a commercial synthesis tool to achieve their area estimate and 2) to eliminate unnecessary features that are not needed in this work from the synthesized model (e.g. violation counter for assertions in confirmation experiments). The slight improvement in the area overhead is a by-product of the latter point. The Author would like to emphasize that the focus of this work is not on assertion synthesis and generic assertion synthesis, tools such as MBAC, support most features in SVA and PSL, that we do not use and therefore our tool does not support.



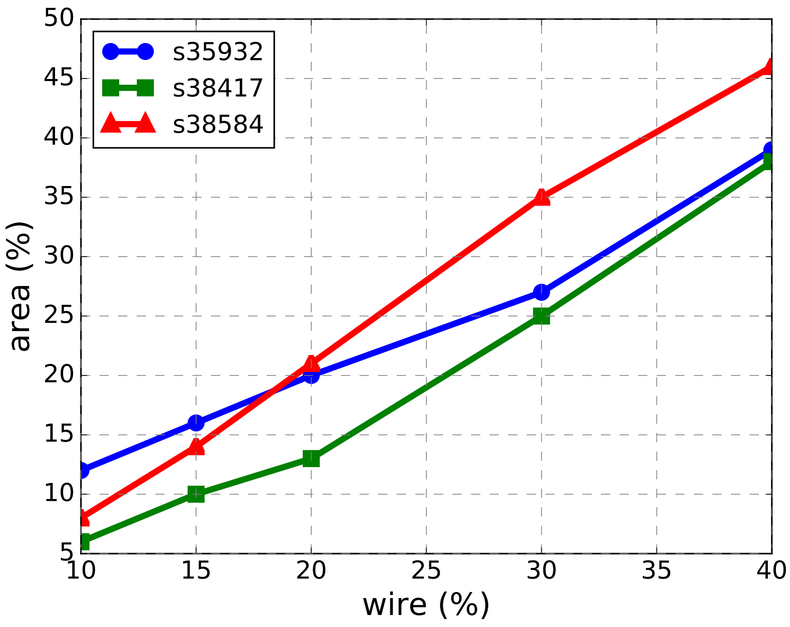
(a) Flip-flop coverage for ISCAS s35932



(b) Flip-flop coverage for ISCAS s38417



(c) Flip-flop coverage for ISCAS s38584



(d) Area overhead with respect to different wire budgets. The ranker is set to maximize flip-flop coverage

Figure 4.16: Evaluation of the flip-flop coverage and the resulting area overhead in confirmation experiments for different wire counts.

4.4.3 Run-times for different steps of our methodology

Finally, we will provide the run-times of each box of the methodology in Figure 4.1. Results provided in Table 4.5 are for the case where ranker has been constraint with 40% wire usage for ISCAS s38584 where the total time spent for finding assertions as well as the total number of assertions is the largest (worst-case scenario). As for assertion generation, GoldMine has been configured to generate assertions both by coverage miner engine and decision forest miner engine. A combination of random sxtimuli as well as deterministic stimuli produced by [150] has been used and the formal verifier has been set to provide 6 to 10 counter examples which are fed back to the miner for refining the original trace. For preparation experiments and confirmation experiments, a total of 28365 and 633 assertions were assessed respectively. As for assertion mapper, the reported run-times are the sum of the run-time for as-

Task	**Configuration**	**Time (hour)**
Assertion Generation	Miner (Single Core), Formal Verifier (Multi Core)	228
Preparation Experiments	FPGA (45 sessions), 256 injections	74
Assertion Mapper	Single Core	0.17
Assertion Ranker	Single Core	0.04
Confirmation Experiments	FPGA (1 session), 5000 injections	16.2

Table 4.5: Evaluation of the running-time of different steps of the tool flow in Figure 1 for ISCAS s38584.

sertion synthesis using our proposed algorithm in section 4.2 as well as the run-time for Synopsys Design Compiler for measuring the area overhead for the final selected assertions.

4.5 Concluding remarks

In this chapter, we have presented a fully automated methodology that generates emulation-ready architectures that can be used to aid the selection and assessment of hardware assertions for bit-flip detection in post-silicon validation. We have shown that, by increasing the number of error injections, the flip-flop coverage estimate is improved up to 10% with shorter run-times compared to that of simulation-based experiments (under the same constraints for wire count). This improvement is largely due to emulation facilitating the discovery of more accurate relationships between assertions and flip-flops that can be covered by them.

The coverage estimates that we obtain are under the assumption that *all* the flip-flops are equally likely to be affected by bit-flips. Therefore, with a wire count constraint of 40% (and an area overhead in the similar range) we are able to cover up to approx 50% of the flip-flops. If the number of flip-flops that need to be monitored is lower (e.g., the destination flip-flops on critical paths) then with similar area and wire count constraint, a significantly improved coverage of the concerned flip-flops will be achieved using our methodology.

Chapter 5

SAT-based Methodology for Designing Bit-flip Detectors

In chapters 3 and 4 we introduced automated methodologies for design of embedded bit-flip detectors. Chapter 3 introduced a complete tool flow and its speed and accuracy limitations were addressed in chapter 4 through the introduction of an emulation architecture. In this chapter, we aim to address another limitation of the methodology in chapter 3 by improving the quality of the initial pool of the generated hardware invariants by introducing our proposed SAT-based invariant generation tool. In addition, we will leverage the potential of the incremental SAT for fast assessment of the generated invariants for bit-flip detection, hence replacing the experimental evaluation of invariants with an accurate formal approach.

Our proposed method is shown in Figure 5.1. The rest of the chapter is organized as follows. Section 5.1 reviews the Boolean Satisfiability Problem (SAT) and provides a brief overview of the internals of the modern SAT-solvers. Section 5.2 discusses our proposed method for generation of hardware invariants that are based on the structural representation of the design (netlist). Afterwards in section 5.4, we will introduce our fully formal approach towards the assessment of the generated invariants based on their potential towards detecting bit-flip. Our method eliminates the need for any form of simulation or emulation experiments. Section 5.5 quickly discusses the ranking algorithm used in this chapter by highlighting the slight modifications from the one used in previous chapters. Finally in section 5.6, we will provide results that indicate clear improvements in flip-flop coverage and the resulting area overhead of the subset of assertions that are selected based on the same constraint as before, yet generated through our own invariant generation engine.

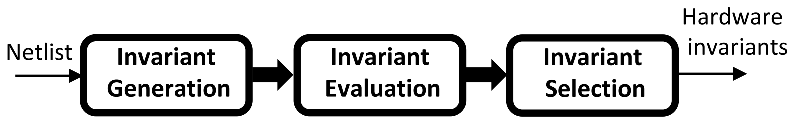


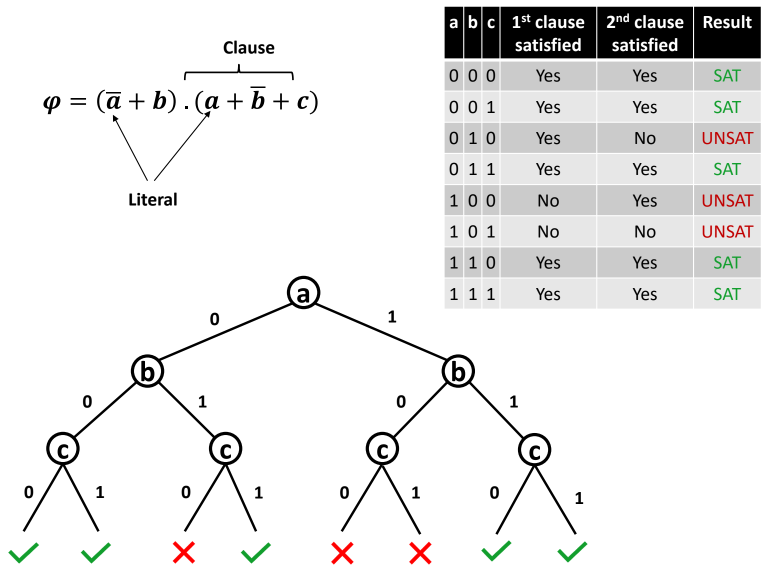
Figure 5.1: Three steps of the proposed SAT-based methodology.

5.1 SAT Fundamentals

Before elaborating on the main steps of our SAT-based methodology that produces hardware invariants suitable for bit-flip detection (illustrated in Figure 5.1), we first provide a brief overview of the Boolean SAT problem.

Although the Boolean SAT problem is known to be a fundamental NP-complete problem [168], there have been significant improvements in recent decades in the development of efficient SAT solvers that enable solving practical instances in electronic design automation (EDA) [169, 170]. The Boolean SAT problem can be summarized as follows.

A conjunctive normal form (CNF) formula φ of n variables $x_0, ..., x_{n-1}$ is the



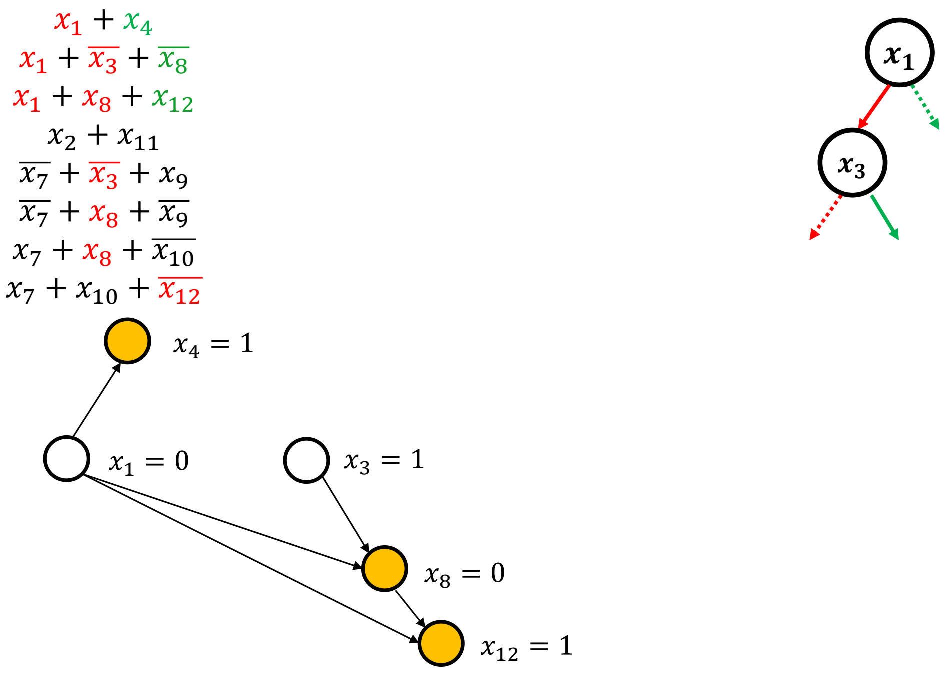
a	b	c	1st clause satisfied	2nd clause satisfied	Result
0	0	0	Yes	Yes	SAT
0	0	1	Yes	Yes	SAT
0	1	0	Yes	No	UNSAT
0	1	1	Yes	Yes	SAT
1	0	0	No	Yes	UNSAT
1	0	1	No	No	UNSAT
1	1	0	Yes	Yes	SAT
1	1	1	Yes	Yes	SAT

Figure 5.2: Example of all possible assignments to a SAT instance.

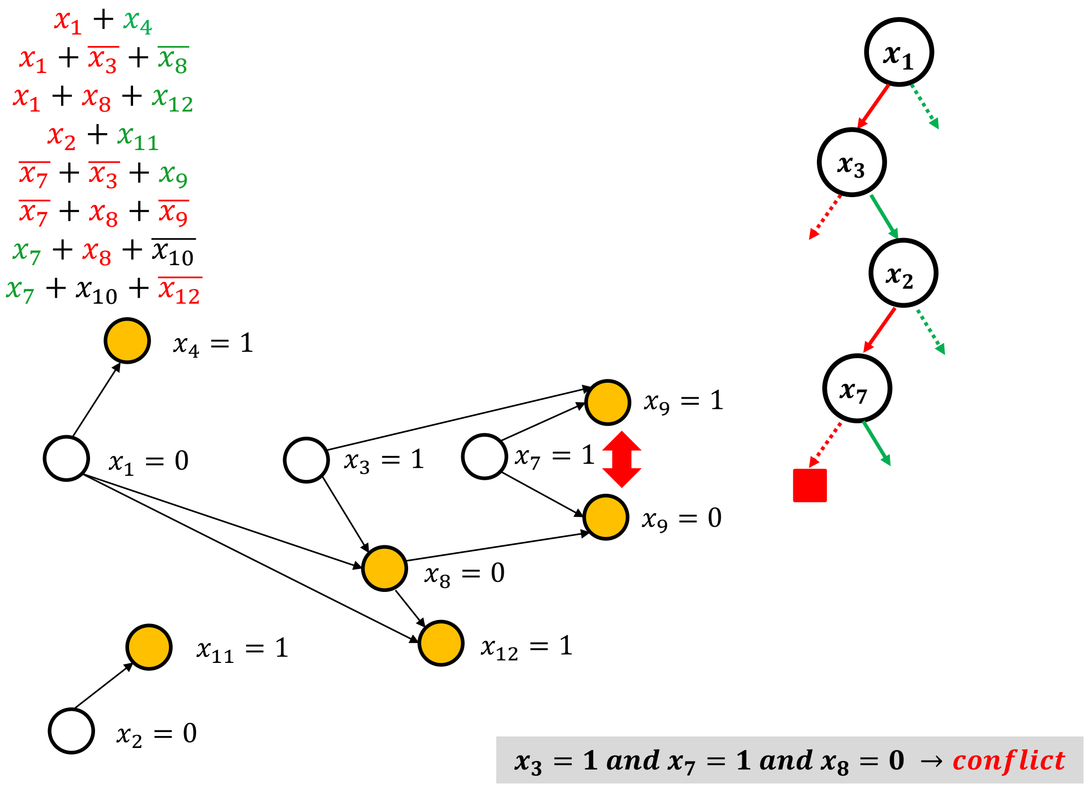
conjunction of clauses $\omega_0, ..., \omega_{m-1}$ where each clause is the disjunction of one or more literals, where each literal is a Boolean variable x_i or its complement $\overline{x_i}$. A clause is satisfied if at least one of its literals evaluates to TRUE. A SAT solver searches for an assignment to the variables such that the CNF formula (SAT instance) evaluates to TRUE (SAT) or proves that such assignment does not exist (UNSAT). For a SAT instance of n variables, there exist 2^n possible true assignments that need be checked. For instance, all possible assignments for the following SAT instance $\varphi = (\bar{a}+b)(a+\bar{b}+c)$ are shown in Figure 5.2. As it can be seen, for this SAT problem with 3 variables, there are 8 possible assignments that need to be checked. As it can be seen, for three of the 8 assignments, the assignment is UNSAT meaning there exists at least one clause which is not satisfied. Please note that, conjunction and disjunction are represented with $\wedge$ and $\vee$ respectively [171]. However, for the sake of simplicity, we represent conjunction (logical AND) using "." and the disjunction (logical OR) using "+".

SAT solvers have evolved significantly over the past few years and are currently used as the core of many EDA tools in verification [172, 173], logic synthesis [174, 175, 176], manufacturing test [177, 178, 179] and so on. There are many algorithms for solving SAT instances such as DPLL [180], using Boolean decision diagrams (BDDs) to solve SAT [181], GSAT and WSAT [182], Stålmarcks algorithm [183] and GRASP [169]. Discussing the algorithmic advances of each of these methods is beyond the scope of this work. Nonetheless, it is worth summarizing the concept of *learned clauses* in Conflict Driven Clause Learning (CDCL) [169, 170, 30] as it is widely used in most of the state-of-the-art SAT solvers [184, 185] and is central to our invariant generation methodology which will be detailed in section 5.2.

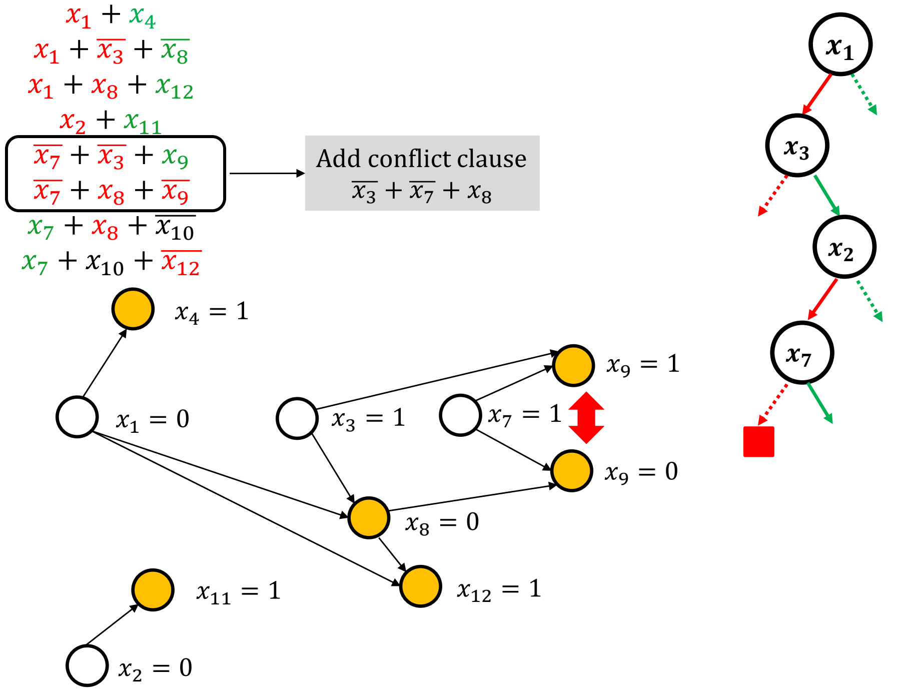
Whenever a partial assignment of Boolean variables $x_i, ..., x_j$ leads to an unsatisfiable clause ω_k, the search engine encounters a conflict and it needs to backtrack. Since not all the assigned variables contribute to the conflict, in order to prune the search space by avoiding the repetitive traversal of the space covered by the partial assignment that lead to the conflict, the reason of the conflict is recorded as a learned clause. For instance, the example shown in Figure 5.3 which is based on [9] represents steps that are taken by a SAT solver that uses CDCL algorithm to solve a SAT instance that is shown as a set of clauses at the left side of the figure. At first step, x_1 is set to 0 which implies x_4 to be set to 1 in order to satisfy the first clause as shown in Figure 5.3a. Note that implied assignments are differentiated by yellow circles. Then x_3 is set to 1 which together with the assignment to x_1 imply x_8 to be 0 to satisfy the second clause. Implied assignment on x_8 together with the explicit assignment on x_1 further implies x_{12} to be set to 1 as shown in Figure 5.3a. Following to this, x_2 is assigned to 0 which implies x_{11} to be 1 to satisfy the fourth clause as shown in Figure 5.3b. Afterwards, x_7 is set to 1 which point implies x_9 to


$x_1 + x_4$
$x_1 + \overline{x_3} + \overline{x_8}$
$x_1 + x_8 + x_{12}$
$x_2 + x_{11}$
$\overline{x_7} + \overline{x_3} + x_9$
$\overline{x_7} + x_8 + \overline{x_9}$
$x_7 + x_8 + \overline{x_{10}}$
$x_7 + x_{10} + \overline{x_{12}}$
$x_4 = 1$
$x_1 = 0$
$x_3 = 1$
$x_8 = 0$
$x_{12} = 1$
x_1
x_3

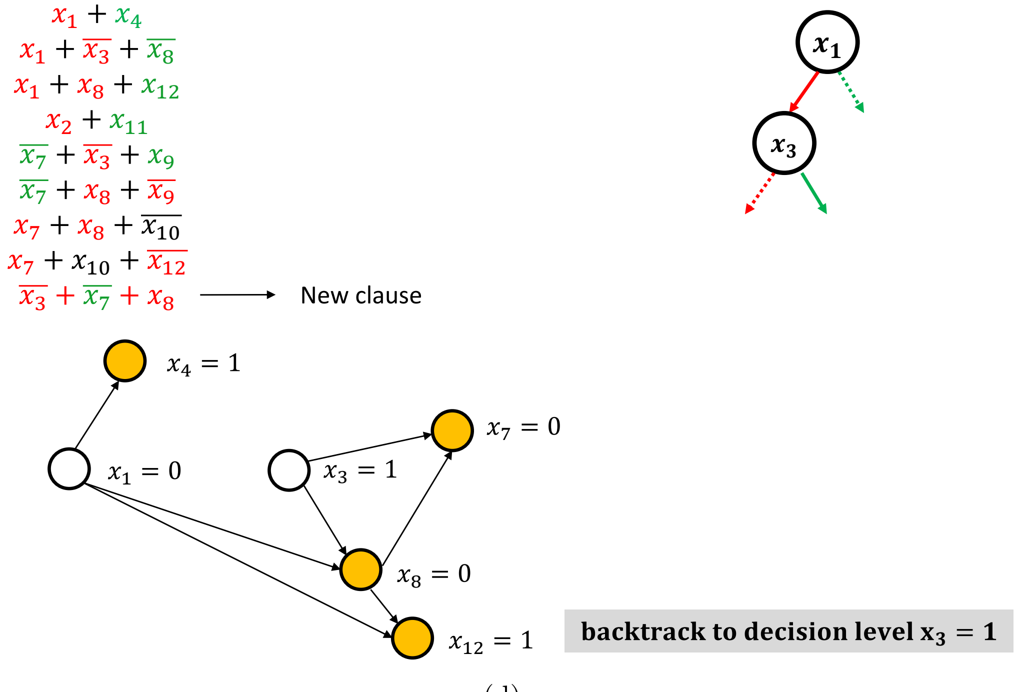
(a)


$x_1 + x_4$
$x_1 + \overline{x_3} + \overline{x_8}$
$x_1 + x_8 + x_{12}$
$x_2 + x_{11}$
$\overline{x_7} + \overline{x_3} + x_9$
$\overline{x_7} + x_8 + \overline{x_9}$
$x_7 + x_8 + \overline{x_{10}}$
$x_7 + x_{10} + \overline{x_{12}}$
$x_4 = 1$
$x_1 = 0$
$x_3 = 1$
$x_7 = 1$
$x_9 = 1$
$x_9 = 0$
$x_8 = 0$
$x_{12} = 1$
$x_{11} = 1$
$x_2 = 0$
x_1
x_3
x_2
x_7
$x_3 = 1$ and $x_7 = 1$ and $x_8 = 0$ → conflict

(b)


$x_1 + x_4$
$x_1 + \overline{x_3} + \overline{x_8}$
$x_1 + x_8 + x_{12}$
$x_2 + x_{11}$
$\overline{x_7} + \overline{x_3} + x_9$
$\overline{x_7} + x_8 + \overline{x_9}$
$x_7 + x_8 + \overline{x_{10}}$
$x_7 + x_{10} + \overline{x_{12}}$
Add conflict clause
$\overline{x_3} + \overline{x_7} + x_8$
$x_4 = 1$
$x_1 = 0$
$x_3 = 1$
$x_7 = 1$
$x_9 = 1$
$x_9 = 0$
$x_8 = 0$
$x_{11} = 1$
$x_{12} = 1$
$x_2 = 0$
x_1
x_3
x_2
x_7

(c)


$x_1 + x_4$
$x_1 + \overline{x_3} + \overline{x_8}$
$x_1 + x_8 + x_{12}$
$x_2 + x_{11}$
$\overline{x_7} + \overline{x_3} + x_9$
$\overline{x_7} + x_8 + \overline{x_9}$
$x_7 + x_8 + \overline{x_{10}}$
$x_7 + x_{10} + \overline{x_{12}}$
$\overline{x_3} + \overline{x_7} + x_8$
New clause
$x_4 = 1$
$x_1 = 0$
$x_3 = 1$
$x_7 = 0$
$x_8 = 0$
$x_{12} = 1$
backtrack to decision level $x_3 = 1$
x_1
x_3

(d)

Figure 5.3: An example showing steps of a CDCL algorithm reproduced from [9]. Implied assignments are shown with yellow circles.

be 1 to satisfy the fifth clause. At this point, a *conflict* is reached because the sixth clause cannot be satisfied as shown in Figure 5.3b. The variables that contributed to the conflict are evaluated to create a new clause which has been *learned* based on the previous explicit and implied assignments as shown in Figure 5.3c. The *learned* clause is added to the clause database and the solver backtracks to the first decision level on the variables that contributed to the conflict which in the case of this example is x_3 as shown in Figure 5.3d. The newly learned clause implies an assignment on x_7 to be 0 hence avoiding the same conflict to be reached again as shown in Figure 5.3d. The next steps for finding whether the problem is `SAT` or `UNSAT` is omitted as it is out of the scope of this discussion. The important points to note are: 1) The *learned clause* can be added to the set of original clauses and 2) it significantly prunes the search space, avoids the generation of the same conflict and helps generating future learned clauses.

Our key observation is that a subset of the learned clauses, which are produced by a SAT solver for a SAT instance based on circuit netlist, are design invariants that can be translated into hardware checkers that can monitor for bit-flips during post-silicon validation. Therefore, the judicious generation, evaluation and selection of these hardware invariants can contribute to the design of more effective bit-flip detectors, as substantiated by our results.

5.2 Generation of Hardware Invariants

Before discussing our method for generating a large pool of invariants through a SAT solver, let us discuss how the conversion of the circuit netlist structure to CNF formula is done through Tseytin transformations. The CNF formula of a combinational circuit

Gate	Function	CNF formula
n-input AND	$x = AND(w_1, \ldots, w_n)$	$[\prod_{i=1}^{n}(w_i + \bar{x})].[(\sum_{i=1}^{j} \bar{w}_i) + x]$
n-input NAND	$x = NAND(w_1, \ldots, w_n)$	$[\prod_{i=1}^{n}(w_i + x)].[(\sum_{i=1}^{j} \bar{w}_i) + \bar{x})]$
n-input OR	$x = OR(w_1, \ldots, w_n)$	$[\prod_{i=1}^{n}(\bar{w}_i + x)].[(\sum_{i=1}^{j} w_i) + \bar{x})]$
n-input NOR	$x = NOR(w_1, \ldots, w_n)$	$[\prod_{i=1}^{n}(\bar{w}_i + \bar{x})].[(\sum_{i=1}^{j} w_i) + x)]$
NOT	$x = NOT(w_1)$	$(x + w_1).(\bar{x} + \bar{w}_1)$
BUFFER	$x = BUFFER(w_1)$	$(\bar{x} + w_1).(x + \bar{w}_1)$

Table 5.1: Tseytin transformations for generating CNF formulas for combinational gates [30].

is the conjunction of each individual gate which is shown in Table 5.1 [30].

Now let us consider an illustrative example. The circuit from the top-left corner of Figure 5.4 can be translated into a CNF formula using Tseytin transformations shown in Table 5.1. For instance, the two-input AND gate with a and b as inputs and c as output is represented using three clauses $(a+\overline{c}).(b+\overline{c}).(\overline{a}+\overline{b}+c)$. Flip-flops are treated as buffers with its input and output in two consecutive clock-cycles (or time-steps), namely $(\overline{D_k}+Q_{k+1}).(D_k+\overline{Q_{k+1}})$, where D_k and Q_{k+1} are Boolean variables for the data input and data output terminals of the D flip-flop in time-steps k and $k+1$ respectively. For example, for the circuit from Figure 5.4, we have $(\overline{c_0}+e_1).(c_0+\overline{e_1})$. The CNF formula for a single time-step is shown in the top-right corner of Figure

5.4. If the circuit is unrolled for multiple clock cycles, the same principle applies by time-shifting, i.e., we add time-step k to the index of each Boolean variable from Figure 5.4.

When a SAT solver prunes the search space through conflict-driven clause learning, it can identify a learned clause like the one shown in the bottom-right corner of Figure 5.4. The clause $(\overline{e_1}+f_1)$ states that e and f cannot be 1 and 0 respectively in the same clock cycle since this would lead to a conflict (note, through time-shifting $(\overline{e_k}+f_k)$ would hold for any positive k). This learned clause is an invariant and it can be implemented in hardware using an OR gate, as shown in the bottom-right corner. An important observation is that if this OR gate is attached to the original circuit, it is guaranteed to detect bit-flips from state 00 to 10 or from 11 to 10. Hence using a single gate, one can ensure that bit-flips in either of the flip-flops are detectable.

As a side note, the logic implications captured by the above-discussed invariant can be captured also using the following two SystemVerilog assertions [27]: $(e ==$

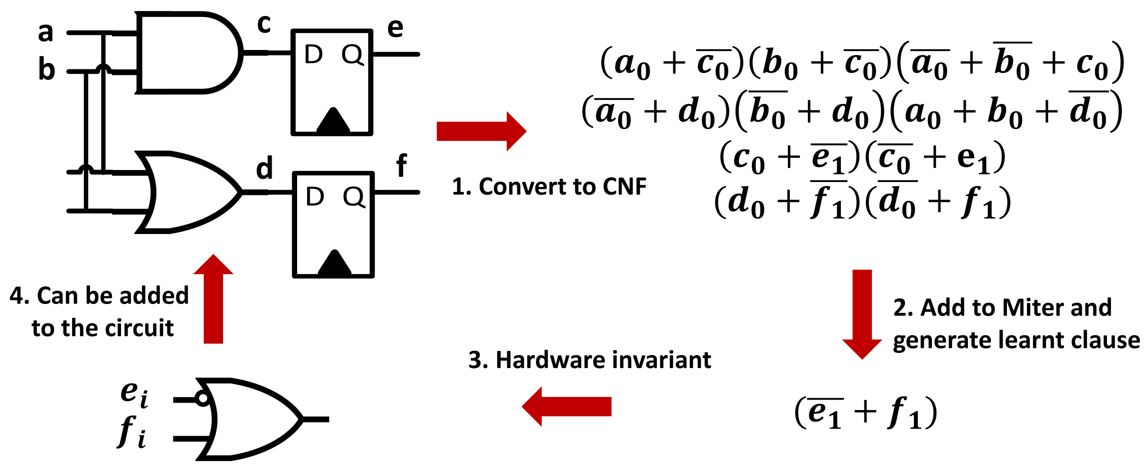

a
b
c
d
e
f
D Q
D Q
1. Convert to CNF
$(a_0+\overline{c_0})(b_0+\overline{c_0})(\overline{a_0}+\overline{b_0}+c_0)$
$(\overline{a_0}+d_0)(\overline{b_0}+d_0)(a_0+b_0+\overline{d_0})$
$(c_0+\overline{e_1})(\overline{c_0}+e_1)$
$(d_0+\overline{f_1})(\overline{d_0}+f_1)$
2. Add to Miter and generate learnt clause
$(\overline{e_1}+f_1)$
3. Hardware invariant
e_i
f_i
4. Can be added to the circuit

Figure 5.4: An example circuit used to illustrate the different sub-steps in hardware invariant generation.

1) |–> ($f == 1$) and ($f == 0$) |–> ($e == 0$). If these two assertions are mapped to hardware, the resulting circuit would be the same OR gate from Figure 5.4. Nonetheless, the point that is important to articulate is that a learned clause has an intuitive and direct translation to hardware through an OR gate between its literals, where each literal would map to a signal from the circuit. A slightly more complex example is shown in Figure 5.6. For the clause $(\overline{a_0} + b_1 + \overline{c_2})$, the hardware invariant is shown below the clause (note Figure 5.6 will be discussed in detail in section 5.4). If the time-steps of the literals are different, shift-registers of depth equal to the time difference between the time-step of the corresponding literal and the highest time-step are introduced at the input of the OR gate. Finally, it is also worth noting that the example from Figure 5.4 is used only for the purpose of explaining the concept of how a learned clause can produce a hardware-implementable design invariant that can capture bit-flips; from the practical standpoint it is unlikely that a SAT solver would need to generate a learned clause for a circuit of such complexity.

Having established that learned clauses produced by SAT solvers (for SAT instances based on circuit netlists) can produce invariants that can be mapped to hardware to monitor for bit-flips, the important question is: "how to make a SAT solver produce a large pool of learned clauses for a circuit?". If a circuit is translated into a CNF like in Figure 5.4, this SAT instance would be immediately satisfiable, thus no significant learned clause discovery process will be undertaken. To address this concern, we build a SAT instance using the miter concept commonly employed in equivalence checking [186]. The upper-side in Figure 5.5 consists of the unrolled original netlist. The lower-side contains another gate-level description that is functionally equivalent but structurally different from the original netlist. While the SAT solver searches for an answer to the SAT instance that constrains the output of the OR gate to `TRUE`, it generates learned clauses that capture non-obvious logic implications, most commonly across multiple clock cycles. Nevertheless, unless special considerations are taken into account, even for miter-type SAT instances an efficient SAT solver would return `UNSAT` rapidly. Hence, it is important to account for the following:

- The two netlists used in the miter must be structurally different, which implies that the original netlist (for which we need to record the learned clauses in order to produce hardware invariant candidates for bit-flip detection) should be re-synthesized using a dissimilar target library or a different optimization objective;

- The netlist should be unrolled for a sufficiently large number of time-steps. This does not only make the SAT solver to reason for a longer time, hence produce more learned clauses that capture non-obvious relationships between

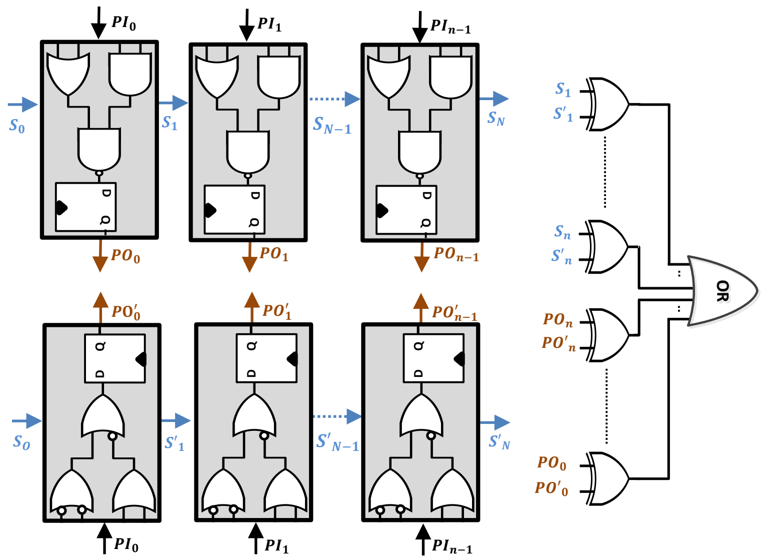


Figure 5.5: Sequential miter configuration used to constrain the SAT solver to produce learned clauses for an extensive period of time.

the circuit's signals, but it also helps discover invariants with a greater temporal depth. As will be discussed in the section 5.4, such invariants can monitor more flip-flops for bit-flips.

- Modern SAT solvers actively manage the learned clause database and frequently discard the learned clauses that did not contribute recently for pruning the search space [184, 185]. Our objective is rather different because we wish to record *all* the learned clauses whose literals map to the original netlist that will be implemented in hardware. Therefore, the learned clauses must be logged as they are generated before the SAT solver discards a portion of them. Nevertheless, due to the miter-based SAT formulation, many learned clauses will capture relationships between the upper and lower-side of the miter, as well as the XOR network that forces the outputs and the intermediate states to be equivalent. Our preliminary experiments have shown that for a circuit with 1,500 flip-flops that is unrolled for 20 clock cycle, over 1 million learned clauses are generated in one hour. Three quarters of the clauses contain literals that refer to signals not in the original netlist and can therefore be discarded.

- Because of learned clause management that most SAT solvers use internally, some clauses that were discarded might be re-discovered at a later point. Since they relate to the same signal dependencies in the circuit, a rediscovered learned clause does not need to be recorded as a hardware invariant candidate that needs to be assessed for bit-flip detectability (see section 5.4 for specific details). Moreover, learned clauses that are time-shifted versions of each other (see discussion for the circuit from Figure 5.4) are also marked as duplicates. Our empirical observation indicates that about 57% of the learned clauses fall into this category, especially those of small temporal depth.

- Another important empirical observation is that a significant amount of clause-learning is done in the first few seconds of a SAT run. In addition, by letting the SAT solver work until it returns `UNSAT`, there is a risk of having localized learned clauses that do not explore signal dependencies in all the regions of the netlist. Hence, instead of running a single long-duration SAT instance, a large number of short-duration SAT instances (using a time-out feature) are run with different initial seeds for random decisions. Furthermore, in order to avoid localized learned clauses, we also prioritize random decision making rather than activity-based decision heuristics, such as variable state independent decaying sum [170]. While our decision might appear counter-intuitive, it is driven by our objective to record a large pool of learned clauses spread across all regions of the circuit, which is fairly different from the common usage of SAT solvers.

5.3 Converting Learned Clauses to SVA Assertions

As explained in the previous section, a learned clause has a direct hardware implication by introducing an OR gate in between the signals that comprise it as shown in Figure 5.4 and ensuring that the signal's time-steps have been updated by incorporating shift-register structures as shown in Figure 5.6. For post-silicon validation, this direct hardware realization is of significant potential because as it will be discussed later, a single learned clause can monitor multiple bit-flips. In addition, as it will also be elaborated in this section, a single learned clause can be translated into multiple assertions meaning that using learned clauses instead of assertions would result in less area overhead. Nevertheless, our proposed method for invariant generation can also be used in pre-silicon verification. For instance, the assertions that are generated from the learned clauses can be used to assess the quality of the applied test vectors during a regression suit. Hence, in this section, we will explain how a learned clause can be translated to an assertion. Please note that SVA [20] will be the assertion language that we will use. Nonetheless, the same principles can be applied for PSL [187] as well.

As it has been explained in several places in this thesis, assertions ensure that design properties are satisfied. There are two types of properties: *safety properties* and *liveness properties* [27]. A safety property which is also known as an invariant describes a behavior that must be hold true for all sample points of time. For instance, the following SVA assertion, $(a == 1)$ && $(b == 0)|=>$ ##1 $(c == 1)$ is a safety property that says if a is 1 and b is 0, c needs to be 1 in two clock-cycles. On the other hand, a liveness property specifies that something should eventually happen. For instance, whenever `req` signal is asserted, the `ack` signal must be asserted sometimes in the future. The learned clauses generated by our flow can be translated into safety properties which are also known as invariants.

Now let us consider the learned clause in Figure 5.4, $\bar{e_1} + f_1$. The subscripts represent the time-step of the signal that is in the learned clause. Since as explained in section 5.2, a learned clause can be added to the original set of clauses and needs to evaluate to 1 at all times, for this learned clause at least one of the literals must evaluate the learned clause to 1. This means that either f_1 needs to be 1 or $\bar{e}_1$ needs to be 0. Hence, since both signals have equal time-steps, at *all times* if e is 1, then f must be 1 and if f is 0, e must be 0. Thus, the SVA assertions for this learned clause are shown below:

```
        assert property ((@posedge clk) ((e == 1)  |-> (f == 1)))

        assert property ((@posedge  clk) ((f == 0) |-> (e == 0)))
```

As it can be seen, the single learned clause of Figure 5.4 can be translated into two different assertions. Now let us consider a case where signals are in different time-steps. For the learned clause of Figure 5.6, which is $(\bar{a}_0 + b_1 + \bar{c}_2)$ each single signal is in a different time-step. Following the same principle that the learned clause must always evaluate to 1, the associated assertion must ensure that whenever signal a is 1 and signal b is 0 one clock cycle later, signal c must be 0 in two clock cycles (time-step of signal a is the reference time-step). Knowing that delays are shown with cycle delay constructs followed by the number of clock cycles or with non-overlapping implication operators, as explained in section 4.2, the SVA assertion for this learned clause will then be:

```
assert property ((@posedge clk) ((a == 1) ##1  (b == 0)  |=> (c == 0)))
```

As it can be seen, there is only one assertion associated with this clause. Adding another signal to the time-step, the associated SVA assertions for the learned clause $(\bar{a}_0 + b_1 + \bar{c}_2 + d_2)$ will be:

```
assert property ((@posedge clk)((a==1) ##1 (b==0) ##1 (c==1) |-> (d==1)

assert property ((@posedge clk)((a==1) ##1 (b==0) ##1 (d==0) |-> (c==0)
```

The important observation is that for learned clauses that have k number of signals in their last time-step, there will be k different assertions that can be derived from the learned clause. That is why for the learned clause of Figure 5.4, two assertions were derived whereas for that of Figure 5.6, one assertion was derived. The following steps represent a generic way for converting learned clauses to SVA assertions.

1. Create a list $L = \{l_1, l_2, \ldots, l_n\}$ of literals where l_i represents the $i-$th literal with its polarity. Note that only literals whose time-step are less than the maximum time-step (last time-step) are placed in this list. For instance, for the learned clause of Figure 5.6, the list is $C = \{-a_0, b_1\}$.

2. Create a second list $K_{maxTF} = \{k_1, k_2, \ldots, k_n\}$ of literals where k_i represents the $k-$th literal whose time-step is equal to the maximum time-step. For instance, for the learned clause of Figure 5.6, the list is $K_{maxTF} = \{c_2\}$.

3. Sweep through C and for each literal l_i, if it is negated write $(l_i == 1)$ and if it not negated, write $(l_i == 0)$. The smallest time-step is considered as the reference time-step and while sweeping through the list, if the time-step is increased by N, cycle-delay construct with delay N, `##N` is used.

4. For each literal in K_{maxTF}, pick one literal k_i at a time and apply the same rule explained in number 3 for the remaining literals. Afterwards, use overlapping constructor and write $(k_i == 0)$ if the literal is negated or $(k_i == 1)$ if it is not.

In this section, we elaborated steps that are needed in order to generate SVA assertions from learned clauses that can be used in pre-silicon verification. For the rest of this chapter, we will exclusively focus on the hardware realization of the learned clauses that can be used to aid detecting bit-flips in flip-flops during post-silicon validation.

5.4 Evaluation of Hardware Invariants

Section 5.2 was focused on invariant generation and it has relied on an *unusual usage* of SAT solvers, i.e., forcing an `UNSAT` instance to run for an extensive period of time in order to record as many learned clauses as possible. Nonetheless, the approach does leverage the reasoning power that modern SAT solvers have been engineered with, namely the identification of non-obvious logic implications. This section is focused on the evaluation of the resulting invariants, instead of assessing them in a simulation or emulation environment as in chapters 3 and 4, for their suitability to detect bit-flips in a hardware implementation. It relies on the *typical usage* of SAT solvers, where many SAT queries are performed rapidly using the concept of incremental SAT solving. Nevertheless, in order to process a very large number of SAT queries, there is a need for an effective problem formulation with an algorithmic front-end that should make it feasible to interface our approach to many existing (or emerging) SAT solvers. The core idea is to create a single SAT instance and perform many incremental SAT queries on it. A SAT query checks the outcome of a SAT instance under a set of assumptions. The key point is that the learning process for one query is leveraged for the subsequent queries. Thus the average time spent on new queries decreases because of the powerful decision heuristics and the sharing of the learned clauses across multiple queries during an incremental SAT run.

The pseudocode for the evaluation of invariants for bit-flip detectability is given in Algorithm 2. The inputs to the algorithm are the netlist of the circuit with m flip-flops, the number of n time-steps to unroll the circuit and a list of invariants $\mathbb{L}$ that were generated using the method from section 5.2. It is worth noting that the unroll parameter n must be at least equal to the maximum temporal depth for the invariants from $\mathbb{L}$. As a first step, we create a single SAT instance φ with activation variables that will be assumed to be either `TRUE` or `FALSE` in different SAT queries. The aim of the activation variables is to modify each pair of clauses for a flip-flop i in time-step j $(D_{i,j} + \overline{Q_{i,j+1}})(\overline{D_{i,j}} + Q_{i,j+1})$ into a pair clauses where a a bit-flip is injected: $(D_{i,j} + Q_{i,j+1})(\overline{D_{i,j}} + \overline{Q_{i,j+1}})$, where $D_{i,j}$ $(Q_{i,j})$ are the Boolean variables for the input (output) of flip-flop i in time-step j. We use two types of activation variables: one type for flip-flops and one type for time-steps. The activation variable for injection in flip-flop i is F_i. Following the same line of reasoning, the activation variable for injection in time-step j is T_j . Based on the above, the pair of clauses for a flip-flop i in time-step j is replaced by six clauses as follows:

$$\begin{aligned} (F_i + D_{i,j} + \overline{Q_{i,j+1}})(F_i + \overline{D_{i,j}} + Q_{i,j+1}) \\ (T_j + D_{i,j} + \overline{Q_{i,j+1}})(T_j + \overline{D_{i,j}} + Q_{i,j+1}) \\ (\overline{F_i} + \overline{T_j} + D_{i,j} + Q_{i,j+1})(\overline{F_i} + \overline{T_j} + \overline{D_{i,j+1}} + \overline{Q_{i,j+1}}) \end{aligned}$$

The total number of activation variables is $m + n$. When no bit-flip is injected, i.e., both F_i and T_j are `FALSE`, the last two clauses will be satisfied, and the first two clauses and the middle two clauses are identical and they map onto the pair of clauses for an error-free flip-flop. When F_i and T_j are both `TRUE` then the first four clauses are satisfied and the last two clauses map onto the pair of clauses for a flip-flop where a bit-flip was injected. Note that if F_i is `TRUE` and T_j is `FALSE` the first two clauses and the last two clauses will be satisfied and the middle two clauses will map to $(D_{i,j} + \overline{Q_{i,j+1}})(\overline{D_{i,j}} + Q_{i,j+1})$, i.e., the error-free flip-flop. The same principle applies when F_i is `FALSE` and T_j is `TRUE`.

After creating the SAT instance, where the pair of clauses for each flip-flop in

```
input  : Circuit netlist with $m$ flip-flops
         The number of time-steps $n$ for unrolling
         List $\mathbb{L}$ of invariants based on learned clauses
output: For each invariant $H_k$ from $\mathbb{L}$ there is a list $\mathbb{D}_k$ of flip-flops in which
         bit-flips can be detected

- Create a SAT instance $\varphi$ based on the netlist with activation variables $F_i$ and
  $T_j$ for each flip-flop $i$ with $0 \leq i < m$ and each time-step $j$ with $0 \leq j < n$
- An assumption set $\mathbb{A}$ is initialized with all of the activation variables $F_i$ and
  $T_j$ set to FALSE
foreach invariant $H_k$ from $\mathbb{L}$ do
    - Add all the literals from $H_k$ to $\mathbb{A}$ as FALSE
    - List of flip-flops $\mathbb{D}_k$ covered by invariant $H_k$ is void
    - $d_k$ is the temporal depth of hardware invariant $H_k$
    foreach time-step $j$ with $0 \leq j < d_k$ do
        - Activate $T_j$ by making it TRUE in $\mathbb{A}$
        - Find the candidate flip-flops for injection $\mathbb{C}_j$
        foreach flip-flop $i$ from $\mathbb{C}_j$ do
            - Activate $F_i$ by making it TRUE in $\mathbb{A}$
            - Solve $\varphi$ with the current assumptions $\mathbb{A}$
            - if the result is SAT then invariant $H_k$ can detect a bit-flip in
              flip-flop $i$ in time-step $j$; add flip-flop $i$ to the set of flip-flops $\mathbb{D}_k$
            - Deactivate $F_i$ by making it FALSE in $\mathbb{A}$
        end
        - Deactivate $T_j$ by making it FALSE in $\mathbb{A}$
    end
    - Remove the literals from $H_k$ from $\mathbb{A}$
end
```

Algorithm 2: Pseudocode for evaluating invariants produced in section 5.2 for their potential to detect bit-flips.

each time-step are replaced by six clauses (as explained above), the intuition behind our approach from Algorithm 2 can be explained as follows. Many incremental SAT queries can be performed on a SAT instance under different sets of assumptions. If we try to force a learned clause associated with an invariant H_k to `FALSE` (by passing an assumption where all the literals of H_k are `FALSE`) then the outcome of the SAT run on the error-free circuit is guaranteed to yield `UNSAT`. This is because the learned clause is an invariant that must hold indefinitely. However, if the assumption also forces a bit-flip in flip-flop i in time-step j, by setting the activation variables F_i and T_j to `TRUE`, then the SAT query will yield either `UNSAT` or `SAT`. If the outcome is `UNSAT` then the bit-flip in flip-flop i in time-step j cannot affect the signal dependencies that make H_k to be an invariant. However, if the outcome is `SAT`, not only does H_k guarantee that a bit-flip in flip-flop i is covered, but also by parsing the assignment of the Boolean variables we can obtain the initial state and the primary inputs sequence that will trigger the invariant when mapped to hardware.

Algorithm 2 has three loop levels. Before the first level, in addition to creating the SAT instance φ, we also initialize the set of assumptions $\mathbb{A}$ with all the activation variables F_i and T_j deactivated (i.e., set to `FALSE`). The outermost loop iterates through all the invariants discovered by the method from section 5.2. For each invariant H_k we first expand the assumptions set $\mathbb{A}$ with its literals assigned to `FALSE`. Unless we activate a pair of F_i and T_j variables, the current set of assumptions $\mathbb{A}$ is guaranteed to yield `UNSAT` to the SAT query. Hence, for each time-step j, where j is upper-bounded by the temporal depth of H_k (called d_k in Algorithm 2), we first activate T_j and then we compute the set of candidate flip-flops $\mathbb{C}_j$ where bit-flip injections will be done ($\mathbb{C}_j$ will be explained intuitively with the example from Figure 5.6). Subsequently, in the innermost loop we activate F_i and we incrementally solve φ under the updated set of assumptions $\mathbb{A}$. If the outcome is `SAT` we can update the set of flip-flops $\mathbb{D}_k$ that are covered by H_k. Note, by reducing the number of flip-flops in which to perform bit-flip injections (by computing $\mathbb{C}_j$) we reduce the number of SAT queries from $m \times \sum_{k=0}^{|\mathbb{L}|-1} d_k$ to $\sum_{k=0}^{|\mathbb{L}|-1} \sum_{j=0}^{d_k-1} |\mathbb{C}_j|$, where any $|\mathbb{C}_j|$ is significantly smaller than the flip-flop count m. Finally, it should be noted that the activation

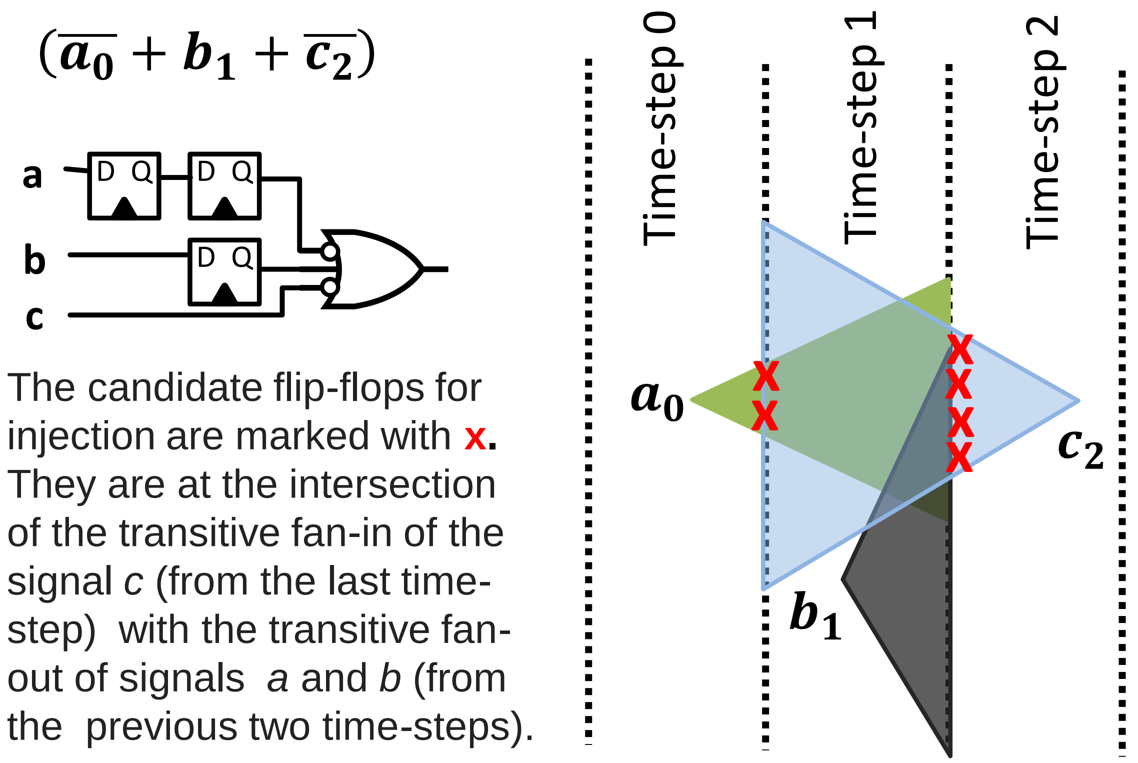

$(\overline{a_0} + b_1 + \overline{c_2})$
a
b
c
D Q
D Q
D Q
The candidate flip-flops for injection are marked with x. They are at the intersection of the transitive fan-in of the signal c (from the last time-step) with the transitive fan-out of signals a and b (from the previous two time-steps).
Time-step 0
Time-step 1
Time-step 2
a_0
b_1
c_2
X

Figure 5.6: An example for showing how to reduce the number of incremental SAT queries during the hardware invariant evaluation for bit-flip detection.

variables F_i and T_j are deactivated at the end of their corresponding loop. Also, before a new invariant is processed, the literals from the previous invariant are removed from the assumptions set $\mathbb{A}$. For each invariant H_k, the algorithm returns the list of flip-flops $\mathbb{D}_k$ that can be covered by the invariant. This set of data can be stored in a sparse matrix format and used by the invariant selection described in section 5.5.

Figure 5.6 shows the intuition behind computing the set of candidate flip-flops $\mathbb{C}_j$ in time-step j. Consider the learned clause $(\overline{a_0} + b_1 + \overline{c_2})$ with a temporal depth of 2. The equivalent hardware implementation of the invariant represented by this learned clause is shown below it. The right-side of the figure illustrates how the set of candidate flip-flops in time-steps 1 and 2 are generated. Through cone-of-influence analysis, by originating from signal c in the last time-step, we compute its transitive fan-in (TFI) [186] recursively until the first time-step is reached. The candidate flip-flops in each time-step must reside in the corresponding TFI. Similarly, for the signals a and b we compute the transitive fan-out (TFO) recursively until reaching the last time-step. The candidate flip-flops in each time-step must reside within the intersection between the TFI of c and the union of TFOs for a and b. It is worth mentioning that the accurate computation of TFO needs to compute the TFI of the respective signal and, thereafter, compute the TFO of the set of signals from this TFI. When applied recursively, this might lead to an increase in runtime while producing a set of candidate flip-flops similar to when computing only the TFI of the signals from the last time-step. Therefore for the results in this work, we rely on the TFI-only method.

5.5 Selection of Hardware Invariants

As fully explained in section 3.2, committing all the generated invariants to the bit-flip detection unit is impractical due to the excessive area overhead. In this section, we discuss how a subset of invariants discovered using the method from section 5.2 can be selected using the set of lists of covered flip-flops (one list for each invariant) generated using the method from section 5.4.

Since the very purpose of finding hardware invariants for post-silicon validation is to embed them to hardware to do on-line monitoring, the equivalent hardware circuits of the discovered learned clauses must be generated. As shown in Figure 5.4, the equivalent hardware of a learned clause whose signals are in the same time-step is constructed by connecting the signals to an OR gate with a fan-in equal to the number of literals in the learned clause. For a learned clause whose signals are in different time-steps, a shift register of depth equal to the time difference between the time-step of the respective literal and the highest time-step in the invariant are used. Signals with the maximum time-step are connected directly to the OR gate either as-is or through an inverter depending in their polarity as shown in Figure 5.6. The estimated area of each invariant is calculated taking into account the number of flip-flops, inverters, and the number of inputs to the OR gate. Each of these parameters can be scaled based on technology dependent coefficients.

In order to select a subset of invariants from the initial pool, we have implemented the ranking algorithm discussed in section 3.2.2. This ranking algorithm is essentially a set covering problem that, based on user-defined constraints (e.g. wire usage), selects a subset of invariants aiming at maximizing the flip-flop coverage. In our implementation, the relationship between invariants and flip-flops is captured in the lists of covered flip-flops determined by the method from section 5.4. Another notable difference from the way the algorithm was used in section 3.2.2 is that instead of using a third-party synthesis tool to find the hardware equivalent model of an assertion, we have used area estimates based on the simple translation of learned clauses into hardware circuits (explained above). Note, while it is sufficient to proceed with estimates for ranking purposes, the area overhead of the selected invariants, reported in the results section, is determined using a commercial synthesis tool.

5.6 Results

Our SAT-based method has been implemented and run on an Intel Core i7 and an Intel Xeon E7 machine, and GCC 4.8.4 has been used as the C++ compiler with

Circuit	1×60 **min run**		240×15 **sec run**	
	Wire < 10	No wire constraint	Wire < 10	No wire constraint
s35932	112224	247204	90001	152192
s38417	74733	138813	116678	216398
s38584	96633	204613	110678	185295

Table 5.2: Total number of invariants generated for ISCAS circuits [8] after an hour when only one long SAT run is executed versus when 240 SAT-runs of 15 seconds each are executed.

Bash, Python 2.7 and Tcl 8.5 for scripting. We have used the Glucose 3.0 SAT solver [185] that is based on MiniSAT [184]. The implementation of Glucose [185] has been extended for logging learned clauses and for their assessment, as discussed in section 5.2. The benchmark circuits used in this section have all been unrolled for 20 clock cycles (n in Figure 5.5 and Algorithm 2). In addition, the ABC sequential synthesis tool [186] has been used to extract two different implementations of the same circuit for the sequential miter from Figure 5.5.

We start by discussing the first part of our methodology, namely invariant generation from section 5.2. We have observed that the number of learned clauses that are generated by a SAT solver is at a higher rate in the beginning of the reasoning process. For instance, Table 5.2 represents the number of generated learned clauses after running the SAT solver for an hour for the three largest ISCAS circuits unrolled for 20 clock cycles, first with 1 single SAT run of 60 minutes and then with 240 SAT runs of 15 seconds each. In addition, the number of invariants that have less than 10 number of distinct wires are shown too because as we will discuss later, invariants that have more than 10 wires are discarded for further consideration due to their imposed area overhead. As it can be seen, the number of generated invariants are almost in par. However, there is another benefit in running multiple SAT instances.

Circuit	Duplicates (%)
s35932	69.1 (%)
s38417	49.8 (%)
s38584	51.3 (%)

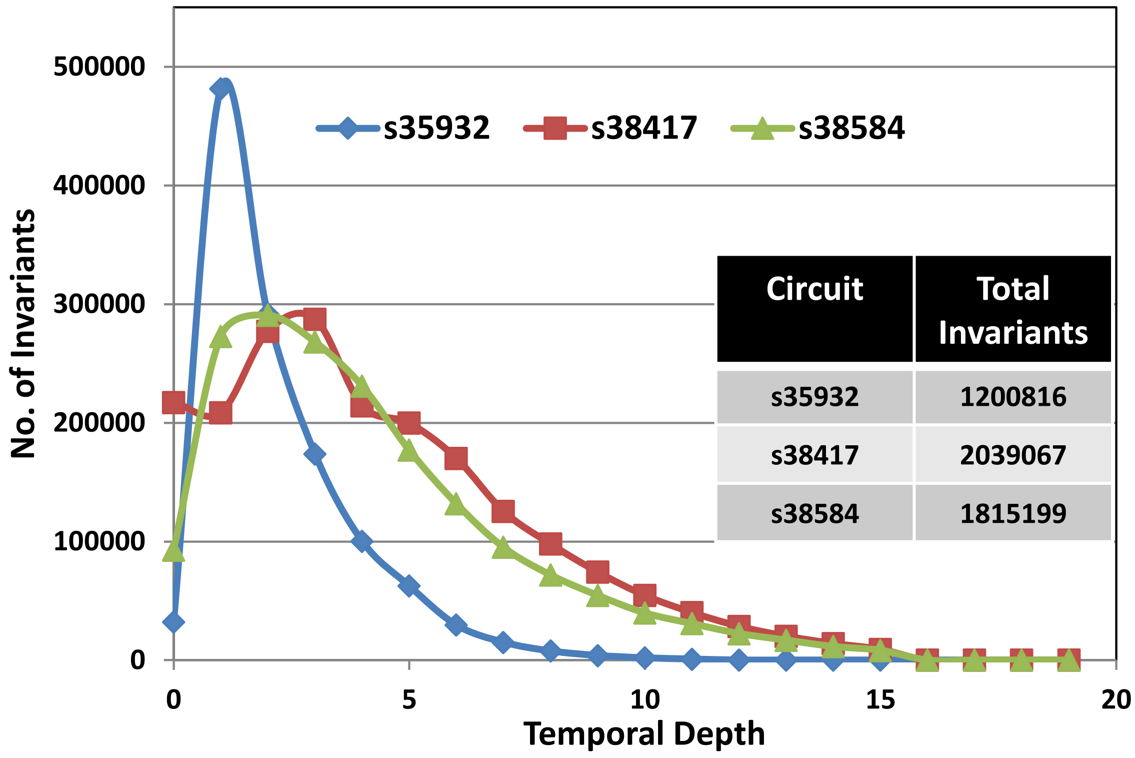
Table 5.3: Percentage of duplicate learned clauses for for running the SAT solver for one hour for each of the three largest ISCAS circuits unrolled for 16 clock cycles without discarding any invariant according to wire constraints.

Re-starting SAT instances with different random seeds will also diversify the generation of invariants to include signals from different regions of the circuit. Hence, we run multiple SAT instances with different random seeds for a predefined time, and subsequently merge the discovered learned clauses ensuring that duplicate clauses are removed. As explained in section 5.2, SAT solvers might find identical relationships between the same signals that are in different time-frames. For instance, the hardware realization of the following two learned clauses $(a_0, \bar{b}_4)$ and $(a_5, \bar{b}_9)$ are identical. Therefore, the duplicate learned clauses must be removed. Table 5.3 represents the percentage of duplicate learned clauses that were found for the three largest ISCAS circuits [8] when 240 SAT-runs of 15 seconds each are run and no learned clause is discarded according to wire constraints (same as last column in Table 5.2). As it can be seen, on average 57% of the learned clauses are duplicates that need to be removed.

In order to prepare a large pool of invariants for assessment, we ran 2,000 SAT instances each for 15 seconds. Figure 5.7 shows the distribution of the total number of distinct invariants after running the method from section 5.2 for a total of 8.3 hours.

To underline the value of SAT-based invariant discovery against simulation-based approaches, we compare against results in chapter 3, which has used GoldMine [3] as their invariant discovery engine. As explained in section 2.3, GoldMine uses data mining on a given simulation snapshot of the circuit and finds the likely invariants that need to pass through a second step (formal verification tool) to remove the false invariants. Referring to table 3.4 in section 3.3.3, the invariant generation part for s38584 [8] takes 228 hours, where 28,365 invariants are discovered with 98.4% of them having a temporal depth below 5. In contrast, the method from this chapter relies on Boolean learning on gate-level netlists, rather than machine learning based on the simulation traces from a behavioral model. Using Boolean satisfiability solvers as the core engine in our work, the need for using third-party formal verification tools is eliminated, thus we have an one-step method. It is important to note that the primary focus of [3] is not on finding invariants that are tuned towards post-silicon validation. Nevertheless, since our method operates on netlists, the benefits of the built-in mechanisms for Boolean learning from within modern SAT solvers are self-evident. Not only that the number of invariants for s38584 is 40 times larger, all of them have been discovered 25 times faster, which amounts to a speed-up of three orders of magnitude per invariant. Furthermore, over 25% of the invariants have a temporal depth greater than 5. Finally, it is worth mentioning that, although it is beyond the scope of this work that is focused on invariants for bit-flip detection during post-silicon validation, it did not escape our attention that the method from section 5.2 can be leveraged for the discovery of invariants that can be used at other steps in the implementation flow, such as identifying hard-to-reach coverage holes during pre-silicon verification.

When evaluating the suitability of each invariant for their potential to detect bit-flips, we have bounded the maximum number of wires for each invariant to 10, in order to limit the area of the bit-flip detection unit. Our experimental results on



Circuit	Total Invariants
s35932	1200816
s38417	2039067
s38584	1815199

Figure 5.7: Total number of discovered invariants as well as their distribution in terms of their temporal depth for the three largest ISCAS circuits.

the three largest ISCAS [8] circuits show that after 32.2 hours on average, between 550,000 to 680,000 invariants are processed in the invariant evaluation part when 4 simultaneous threads are utilized for each circuit. The runtime reported in table 3.4 in section 3.3.3 for assessing the bit-flip detection potential of the discovered invariants in a simulation-based approach for s38584 is 63 hours. Based on our empirical observations, in this work on average more than 100 SAT queries are performed in each second, which validates the efficiency of our incremental SAT formulation from section 5.4, and its suitability to handle a very large number of invariants. Note, the number of SAT queries is dependent on the number of flip-flops that need to be considered for bit-flip injection for each invariant, which is not influenced by the circuit

Wire	**Invariants from chapter 3**			**Invariants from this chapter**		
	s35932	s38417	s38584	s35932	s38417	s38584
10%	22%	12%	19%	43%	31%	30%
15%	30%	25%	28%	56%	37%	38%
20%	42%	28%	34%	67%	42%	45%
30%	62%	39%	47%	79%	53%	56%
40%	77%	48%	59%	86%	64%	66%

Table 5.4: The percentage of flip-flops that are monitored for the three largest ISCAS circuits for the selected invariants from [56] and this work.

size but rather by the invariant depth and the dimensions of the input/output logic cones for the signals from the respective invariant (see Algorithm 2 and Figure 5.6). To summarize, when compared to the simulation-based approach from chapter 3, we can process approx 25 times more invariants in half the time. More importantly, due to the exhaustive nature of the SAT-based approach, we are capable of discovering relationships between invariants and flip-flops that will be missed by simulation, the benefits of which are discussed next.

The advantages of the discovered invariants for post-silicon validation are assessed based on the proportion of flip-flops that can be covered by them when they are embedded on-chip. To select a subset of invariants for hardware implementation, we use the ranking algorithm described in section 3.2.2, with the modifications summarized in section 5.5. The wire usage, which is an important practical constraint for hardware, has been varied from 10% to 40% of the total number of flip-flops. Table 5.4 shows the percentage of flip-flops that are monitored for bit-flips (flip-flop coverage) under different wire constraints. The improvements of this work over the one from chapters 3 and 4 are due to: (i) the initial pool of invariants is significantly larger for this work; (ii) the incremental SAT approach discovers relationships between invariants and bit-flips in flip-flops that cannot be identified through a simulation-based

Wire	**Invariants from [56]**			**This work**		
	s35932	s38417	s38584	s35932	s38417	s38584
10%	12%	6%	7%	5%	8%	6%
15%	16%	10%	18%	8%	11%	10%
20%	19%	14%	21%	12%	15%	14%
30%	27%	25%	35%	19%	20%	21%
40%	38%	37%	46%	26%	27%	28%

Table 5.5: Area overhead of the bit-flip detection unit for the three largest ISCAS circuits for the selected invariants from [56] and this work.

approach. Capturing a stronger relationship between flip-flops and invariants translates into a more accurate selection of the invariants to be mapped to hardware. The results for the area overhead are shown in Table 5.5. For the same underlying reasons as above, in this work we can choose less invariants and/or invariants that can cover more flip-flops.

5.7 Summary

In this chapter we have investigated a novel approach for the generation of invariants and their evaluation for bit-flip detection. While the approach for generation of invariants relies on a counter-intuitive usage of SAT solvers, it does build on the core strengths of these solvers, in particular efficient analysis of the implication graphs for learning new clauses. A large pool of invariants is subsequently evaluated for their potential to detect bit-flips using an effective incremental SAT-based algorithm. The results show a clear improvement over the simulation-based approaches.

Chapter 6

Conclusion

The constant growth in the complexity of SoCs as well as the demand for shorter time-to-market windows have further aggravated the challenge of manufacturing error-free products. Thus, the various steps in the implementation flow have to keep pace with these rapid changes. Post-silicon validation, as the final proof of due diligence in the flow is not an exception in this regard. Existing post-silicon validation practices commonly rely on *ad-hoc* methods that are specific to the target design. Therefore, automatic and systematic methods must be developed in order to minimize the manual intervention in the debugging efforts.

Limited internal observability is one of the most difficult challenges amongst different phases of post-silicon validation. To address this challenge, scan-based debug and embedded logic analysis have been introduced as two complementary approaches as explained in chapter 2. The latter is of particular interest in post-silicon validation because it offers real-time data acquisition without the need to halt the circuit operation. Event detection, as one of the core sub-blocks of the ELAs monitors the circuit and controls trace acquisition when a particular event of interest occurs. One of the directions in the design of state-of-the-art event detection blocks is to use hardware assertion checkers. Hence, in this thesis, we have proposed systematic methods for automated design of assertion-based event detectors. The rest of this chapter is organized as follow. Section 6.1 summarizes the technical contributions of this thesis and section 6.2 discusses possible future works based on the findings from this thesis.

6.1 Summary of thesis contributions

In chapter 3, we have introduced a systematic methodology for automatic generation and selection of assertions based on their potential for finding bit-flips in flip-flops. The proposed methodology generates assertions based on the structural model of the circuit (netlist). Hence, it is fully generic and can be applied to logic blocks of any kind. At first, a large pool of assertions are generated and their potential in detecting bit-flips is found through simulation-based experiments from which the relationship between assertions and flip-flops of the design is captured in a two dimensional matrix. Since instrumenting all the discovered assertions on-chip is practically infeasible, a subset of these assertions are selected through a ranking process which is constrained by the total number of wires used by all the selected assertion and takes into account the information from the two-dimensional matrix that captures the relationship between flip-flops and assertions as well as some attributes of the assertions such as number of wires that comprise each assertion as well as their estimated area overhead. Following to the selection of assertions, their quality is measured by two coverage metrics; bit-flip coverage metric and flip-flop coverage metric as explained in section 3.1.6.

In chapter 4, we improve our proposed methodology in chapter 3 by introducing a fully automated tool flow for designing emulation-ready hardware architectures to replace simulation-based experiments in the proposed methodology of chapter 3. Since post-silicon validation sessions run at silicon speed, efforts should be made to close the gap between simulation and silicon speed. FPGA-based emulators are commonly used to close this gap. Hence, our proposed tool flow generates emulation-ready architectures that can be used to run FPGA-based experiments to assess the potential of assertions for bit-flip detection. This means that the process of assessment and selection of assertions are done more accurately because of a significantly larger number of stimuli that is applied in a much shorter time, hence capturing a more realistic behavior of the circuit.

Finally in chapter 5, we propose our own automatic invariant generation tool which is fundamentally different from the assertion generation tool used in the first two contributions because there is no dependency on simulation traces. In fact, since post-silicon validation is performed when the design has been implemented and verified (netlist is ready), an efficient assertion generation tool should be developed in such way that it relies on the structural relationships between signals without getting feedback from simulation traces. Therefore, our tool reads the netlist, converts it to the CNF formula readable by Boolean Satisfiablity solvers and through formation of a sequential miter circuit, generates hardware invariants that can be used for bit-flip detection. The generated invariants are essentially the learned clauses that are generated as the SAT solver works towards proving that the miter structure is `UNSAT`. By using the same framework, the assessment of the generated invariants can also be evaluated using the incremental SAT solving approach of the modern SAT solvers which eliminates the need for simulation-based or emulation-based experiments to evaluate the potential of every single invariant to detect bit-flips. In addition, a significantly larger number of invariants are generated and assessed in a significantly shorter time period.

6.2 Future research direction

In this section, a few possible future research directions based on the work presented in this thesis are outlined.

As shown in Figure 2.1, post-silicon validation can be divided into many sub-areas from which controllability, observability and root-cause analysis were discussed in chapter 2. The contributions from this thesis are categorized into the observability sub-area where assertions are used as event-detectors to improve internal observability. Nonetheless, once an error is detected, root-cause analysis is started where the ultimate goal is to localize the error and determine why it has occurred. As discussed, this is commonly achieved by analyzing the information recorded in the on-chip trace buffers. Clearly, the success of this step depends on the choice of signals that are selected for which traces are collected. Trace signal selection has been well studied over the past few years [142, 141, 188, 189]. Therefore, an interesting direction of research could be to create a bridge between observability and root-cause analysis by combining the information from the violated assertions with the recorded data on the embedded trace buffers. An assertion that has been violated will provide a specific scenario for which a particular property was not hold. Since signals that comprise each property can reside in different regions of the design, this extra information from the violated assertion can aid further shrink the suspected regions that the error has occurred in. In addition, a specific budget can be allocated to the signals that are connected to the event-detector to be also connected to the trace memories in order to collect their history. This will help identify if the assertion has been sensitized in the clock cycles before the error occurs and if yes, in which iteration it has failed. All these extra information should ideally help pinpointing the most probable regions of the design in which the error could have occurred.

Another possible research direction, is based on the findings from the last contribution of the thesis which was elaborated in chapter 5. As explained, one of the key advantages of our proposed approach for generating hardware invariants is the fact that the non-obvious relationships that span through multiple clock cycles in between signals are found based on the netlist structure. This in particular targets the inherent difference between post-silicon validation and pre-silicon verification in that the focus is on finding susceptible regions of the circuit that are affected by electrical problems. Recall that the potential of the generated invariants based on the method in chapter 5 is assessed based on a formal software-based approach. Therefore, since the invariants are found based on the netlist structure, it is possible that certain invariants that are supposed to detect bit-flips in a particular flip-flop are only excited following to a very particular sequence of stimuli that need to be applied. Therefore, to ensure that the invariants are adequately exercised, the random stimuli need to be constrained to produce stimuli that satisfy the trigerring condition for the invariants. The problem of generating on-chip constrained random stimuli for post-silicon validation has been studied in [77, 78, 190]. Hence, the focus of this research direction must be on analyzing the netlist as well as assertions to provide the necessary information to the work of [77, 78, 190] in order to generate a comprehensive set of constrained stimuli that exercise the silicon prototypes for very long periods of time (on the order of hours or even days). Instead of applying unconstrained randomized patterns, a large number of constrained random stimuli can be applied such that the the circuits functionality is exercised in such way that the enabling conditions of the assertions are adequately exercised.

Assertion-based verification has been used extensively during the pre-silicon verification stages for over a decade. In this dissertation we have studied the suitability of using hardware assertion checkers for bit-flip detection during post-silicon validation. We have proposed a new methodology whose aim is to assess a large pool of assertions and select only a subset of them for post-silicon validation. This methodology, together with all the algorithms and architectures that support it, are generic and can be applied to any digital logic blocks.

Bibliography

[1] David Brock, editor. *Understanding Moore's Law: Four Decades of Innovation.* Chemical Heritage Foundation, 2006.

[2] S. Hangal, S. Narayanan, N. Chandra, and S. Chakravorty. IODINE: a tool to automatically infer dynamic invariants for hardware designs. In *ACM/IEEE Design Automation Conference, 2005. Proceedings. 42nd*, pages 775–778, June 2005.

[3] 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*, 32(6):952–965, June 2013.

[4] H. F. Ko, A. B. Kinsman, and N. Nicolici. Design-for-debug architecture for distributed embedded logic analysis. *IEEE Transactions on Very Large Scale Integrated (VLSI) Systems*, 19(8):1380–1393, Aug 2011.

[5] T. Hong, Y Li, Sung-Boem 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)*, pages 1–10, Nov 2010.

[6] Ming Gao and Kwang-Ting Cheng. A case study of time-multiplexed assertion checking for post-silicon debugging. In *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pages 90–96, June 2010.

[7] J. Geuzebroek and B. Vermeulen. Integration of hardware assertions in systems-on-chip. In *IEEE International Test Conference (ITC)*, pages 1–10, Oct 2008.

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

[9] Sharad Malik. The quest for efficient boolean satisfiability solvers. `http://www.princeton.edu/~sharad/CMUSATSeminar.pdf`. Accessed: 2017-01-20.

[10] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital Integrated Circuits*. Pearson, 2nd edition, 2003.

[11] Paul E. Ceruzzi. *A History of Modern Computing (History of Computing)*. The MIT Press, second edition, 4 2003.

[12] J. Bardeen and W. H. Brattain. The transistor, a semi-conductor triode. *Phys. Rev.*, 74:230–231, Jul 1948.

[13] W. Shockley. The theory of p-n junctions in semiconductors and p-n junction transistors. *Bell System Technical Journal*, 28(3):435–489, 1949.

[14] R. Norman, J. Last, and I. Haas. Solid-state micrologic elements. In *IEEE International Solid-State Circuits Conference (ISSCC)*, volume III, pages 82–83, Feb 1960.

[15] Gordon E. Moore. Readings in computer architecture. chapter Cramming More Components Onto Integrated Circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[16] Zainalabedin Navabi. *VHDL: Analysis and Modeling of Digital Systems*. McGraw-Hill Professional, 2nd edition, 12 1997.

[17] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Science/Engineering/Math, 1nd edition, 1994.

[18] Thorsten Grtker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Springer, 2002 edition, 2002.

[19] S. Palnitkar. *Verilog HDL: A Guide to Digital Design and Synthesis*. Number v. 1 in Verilog HDL: A Guide to Digital Design and Synthesis. SunSoft Press, 2003.

[20] 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 US, 2006.

[21] Malay Ganai and Aarti Gupta. *SAT-Based Scalable Formal Verification Solutions (Integrated Circuits and Systems)*. Springer, 2007 edition, 5 2007.

[22] William K. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Prentice Hall, 1st edition, 2005.

[23] Prakash Rashinkar, Peter Paterson, and Leena Singh. *System-on-a-Chip Verification: Methodology and Techniques*. Springer, 2002 edition, 2013.

[24] S. Devadas, A. Ghosh, and K. Keutzer. An observability-based code coverage metric for functional simulation. In *ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 418–425, Nov 1996.

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

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

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

[28] Lun Li and Mitchell Thornton. *Digital System Verification: A Combined Formal Methods and Simulation Framework (Synthesis Lectures on Digital Circuits and Systems).* Morgan and Claypool Publishers, 2010.

[29] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: A survey. *ACM Transactions in Design Automation of Electronic Systems.*, 4(2):123–193, April 1999.

[30] J. P. Marques-Silva and K. A. Sakallah. Boolean satisfiability in electronic design automation. In *ACM/IEEE Design Automation Conference (DAC)*, pages 675–680, June 2000.

[31] M. Bushnell and Vishwani Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits (Frontiers in Electronic Testing)*. Springer, 2000.

[32] J. H. Patel. Stuck-at fault: a fault model for the next millennium. In *IEEE International Test Conference (ITC)*, pages 1166–, Oct 1998.

[33] S. Chakravarty and V. P. Dabholkar. Two techniques for minimizing power dissipation in scan circuits during test application. In *IEEE Asian Test Symposium (ATS)*, pages 324–329, Nov 1994.

[34] A. S. Mudlapur, V. D. Agrawal, and A. D. Singh. A random access scans architecture to reduce hardware overhead. In *IEEE International Test Conference (ITC)*, pages 9 pp.–358, Nov 2005.

[35] S. Samaranayake, E. Gizdarski, N. Sitchinava, F. Neuveux, R. Kapur, and T. W. Williams. A reconfigurable shared scan-in architecture. In *IEEE VLSI Test Symposium (VTS)*, pages 9–14, April 2003.

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

[37] A Adir, A Nahir, G. Shurek, A Ziv, C. Meissner, and J. Schumann. Leveraging pre-silicon verification resources for the post-silicon validation of the IBM POWER7 processor. In *ACM/IEEE Design Automation Conference (DAC)*, pages 569–574, June 2011.

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

[39] Anand Shimpi. The source of Intel's Cougar Point SATA bug, 2016.

[40] A.B.T. Hopkins and K.D. McDonald-Maier. Debug support for complex Systems-on-Chip: a review. *IEE Proceedings in Computers and Digital Techniques*, 153(4):197–207, July 2006.

[41] B. Vermeulen and S.K. Goel. Design for debug: catching design errors in digital chips. *IEEE Design Test of Computers*, 19(3):35–43, May 2002.

[42] Intel Corp. Intel platform and component validation. `http://download.intel.com/design/chipsets/labtour/PVPT_WhitePaper.pdf`, 2003. [Online; accessed 19-Dec-2016].

[43] A. Evans, A. Silburt, G. Vrckovnik, T. Brown, M. Dufresne, G. Hall, Tung Ho, and Ying Liu. Functional verification of large ASICs. In *ACM/IEEE Design and Automation Conference (DAC)*, pages 650–655, June 1998.

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

[45] R. McLaughlin, S. Venkataraman, and C. Lim. Automated debug of speed path failures using functional tests. In *IEEE VLSI Test Symposium (VTS)*, pages 91–96, May 2009.

[46] J. B. Dugan, S. J. Bavuso, and M. A. Boyd. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Transactions on Reliability*, 41(3):363–377, Sep 1992.

[47] Kai-Hui Chang, I.L. Markov, and V. Bertacco. Automating post-silicon debugging and repair. In *ACM/IEEE Computer-Aided Design*, pages 91–98, Nov 2007.

[48] M. Fujita and H. Yoshida. Post-silicon patching for verification/debugging with high-level models and programmable logic. In *ACM/IEEE Asia and South Pacific esign Automation Conference (ASP-DAC)*, pages 232–237, Jan 2012.

[49] S. B. Park, T. Hong, and S. Mitra. Post-silicon bug localization in processors using instruction footprint recording and analysis (IFRA). *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 28(10):1545–1558, Oct 2009.

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

[51] B. Vermeulen and S.K. Goel. Design for debug: catching design errors in digital chips. *IEEE Design Test of Computers*, 19(3):35–43, May 2002.

[52] Ping Yeung and K. Larsen. Practical assertion-based formal verification for SoC designs. In *International Symposium on System-on-Chip*, pages 58–61, Nov 2005.

[53] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rulke. Advanced verification by automatic property generation. *IET transactions on Computers Digital Techniques*, 3(4):338–353, July 2009.

[54] 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)*, pages 613–620, March 2007.

[55] 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)*, pages 73–78, March 2015.

[56] 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 (TCAD)*, 35(12):2118–2130, 2016.

[57] Kwang-Ting Cheng, Shi-Yu Huang, and Wei-Jin Dai. Fault emulation: A new methodology for fault grading. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 18(10):1487–1495, Oct 1999.

[58] D. Jones and D. M. Lewis. A time-multiplexed FPGA architecture for logic emulation. In *IEEE Custom Integrated Circuits Conference*, pages 495–498, May 1995.

[59] R. Nyberg, J. Nolles, J. Heyszl, D. Rabe, and G. Sigl. Closing the gap between speed and configurability of multi-bit fault emulation environments for security

and safety-critical designs. In *Euromicro Conference on Digital System Design*, pages 114–121, Aug 2014.

[60] Pouya Taatizadeh and Nicola Nicolici. Emulation-based selection and assessment of assertion checkers for post-silicon validation. In *IEEE International Conference on Computer Design (ICCD)*, pages 46–53, Oct 2015.

[61] P. Taatizadeh and N. Nicolici. Emulation infrastructure for the evaluation of hardware assertions for post-silicon validation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems (TVSLI)*, PP(99):1–15, 2017.

[62] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson. GoldMine: Automatic assertion generation using data mining and static analysis. In *ACM/IEEE Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 626–629, March 2010.

[63] L. Liu, D. Sheridan, W. Tuohy, and S. Vasudevan. Towards coverage closure: Using GoldMine assertions for generating design validation stimulus. In *ACM/IEEE Design, Automation Test in Europe (DATE)*, pages 1–6, March 2011.

[64] M. Gopal. *Modern Control System Theory.* Wiley, 1993.

[65] M. G. Bartley, D. Galpin, and T. Blackmore. A comparison of three verification techniques: directed testing, pseudo-random testing and property checking. In *ACM/IEEE Design Automation Conference DAC*, pages 819–823, 2002.

[66] R. Dabic, S. Jednak, I. Adzic, D. Stanic, A. Mijatovic, and S. Vuckovic. Direct

test methodology for HDL verification. In *International Symposium on Design and Diagnostics of Electronic Circuits Systems*, pages 115–118, April 2015.

[67] N. Kitchen and A. Kuehlmann. Stimulus generation for constrained random simulation. In *ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 258–265, Nov 2007.

[68] J. Yuan, C. Pixley, and A. Aziz. *Constraint-Based Verification*. Springer, 2006.

[69] S. Rayadurgam and M. P. E. Heimdahl. Coverage based test-case generation using model checkers. In *IEEE International Conference and Workshop On the Engineering of Computer-Based Systems*, pages 83–91, 2001.

[70] F. Fallah, S. Devadas, and K. Keutzer. OCCOM-efficient computation of observability-based code coverage metrics for functional verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 20(8):1003–1015, Aug 2001.

[71] M. S. Abadir, J. Ferguson, and T. E. Kirkland. Logic design verification via test generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 7(1):138–148, Jan 1988.

[72] I. Iliuta and C. Tepu. Constraint random stimuli and functional coverage on mixed signal verification. In *International Semiconductor Conference (CAS)*, pages 237–240, Oct 2014.

[73] 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 *ACM/IEEE Design Automation Conference (DAC)*, pages 94–95, June 2010.

[74] S. K. Sadasivam, S. Alapati, and V. Mallikarjunan. Test generation approach for post-silicon validation of high end microprocessor. In *Euromicro Conference on Digital System Design*, pages 830–836, Sept 2012.

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

[76] 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 *ACM/IEEE Design, Automation Test in Europe (DATE)*, pages 1–6, March 2011.

[77] 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 (TCAD)*, 35(6):1012–1025, June 2016.

[78] X. Shi and N. Nicolici. Generating cyclic-random sequences in a constrained space for in-system validation. *IEEE Transactions on Computers (TCOMP)*, 65(12):3676–3686, Dec 2016.

[79] 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)*, pages 1–10, Nov 2012.

[80] L.T. Wang, Y.W. Chang, and K.T. Cheng. *Electronic Design Automation: Synthesis, Verification, and Test*. Systems on Silicon. Elsevier Science, 2009.

[81] Claudionor Nunes Coelho and Harry D. Foster. *Assertion-Based Verification*, pages 167–204. Springer US, Boston, MA, 2004.

[82] *Assertion-Based Verification*, pages 167–179. Springer US, Boston, MA, 2007.

[83] N. Nataraj, T. Lundquist, and Ketan Shah. Fault localization using time resolved photon emission and stil waveforms. In *IEEE International Test Conference (ITC)*, volume 1, pages 254–263, Sept 2003.

[84] J. M. Soden and R. E. Anderson. IC failure analysis: techniques and tools for quality reliability improvement. *Proceedings of the IEEE*, 81(5):703–715, May 1993.

[85] M. Paniccia, T. Eiles, V. R. M. Rao, and Wai Mun Yee. Novel optical probing technique for flip chip packaged microprocessors. In *IEEE International Test Conference (ITC)*, pages 740–747, Oct 1998.

[86] R. H. Livengood and D. Medeiros. Design for (physical) debug for silicon microsurgery and probing of flip-chip packaged integrated circuits. In *IEEE International Test Conference 1999*, pages 877–882, 1999.

[87] D. P. Vallett. IC failure analysis: the importance of test and diagnostics. *IEEE Design Test of Computers*, 14(3):76–82, Jul 1997.

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

[89] Ilya Wagner. *Post-Silicon and Runtime Verification for Modern Processors.* Springer Publishing Company, Incorporated, 2014.

[90] K. H. Chang, V. Bertacco, and I. L. Markov. Simulation-based bug trace minimization with bmc-based refinement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 26(1):152–165, Jan 2007.

[91] A. M. Gharehbaghi and M. Fujita. Formal verification guided automatic design error diagnosis and correction of complex processors. In *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pages 121–127, Nov 2011.

[92] L. Xie, A. Davoodi, and K. K. Saluja. Post-silicon diagnosis of segments of failing speedpaths due to manufacturing variations. In *ACM/IEEE Design Automation Conference*, pages 274–279, June 2010.

[93] B. Le, D. Sengupta, A. Veneris, and Z. Poulos. Accelerating post silicon debug of deep electrical faults. In *IEEE International On-Line Testing Symposium (IOLTS)*, pages 61–66, July 2013.

[94] Cadence. *Assertion Writing Guide.* Cadence Design Systems.

[95] M. Fujita. Automatic identification of assertions and invariants with small numbers of test vectors. In *IEEE International Conference on Computer Design (ICCD)*, pages 463–466, Oct 2015.

[96] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rulke. Automatic generation of complex properties for hardware designs. In *ACM/IEEE Design, Automation and Test Conference in Europe (DATE)*, pages 545–548, March 2008.

[97] Lintao Zhang and S. Malik. Conflict driven learning in a quantified boolean satisfiability solver. In *ACM/IEEE International Conference on Computer Aided Design (ICCAD)*, pages 442–449, Nov 2002.

[98] J. Whittemore, J. Kim, and K. Sakallah. Satire: A new incremental satisfiability engine. In *ACM/IEEE Design Automation Conference (DAC)*, pages 542–545, June 2001.

[99] H. Mangassarian, H. Yoshida, A. Veneris, S. Yamashita, and M. Fujita. On error tolerance and engineering change with partially programmable circuits. In *ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 695–700, Jan 2012.

[100] Cadence Design Systems. `http://www.cadence.com`. 2015.

[101] Xinli Gu, Weili Wang, K. Li, Heon Kim, and S. S. Chung. Re-using DFT logic for functional and silicon debugging test. In *IEEE International Test Conference (ITC)*, pages 648–656, 2002.

[102] R. Datta, A. Sebastine, and J. A. Abraham. Delay fault testing and silicon debug using scan chains. In *IEEE European Test Symposium (ETS)*, pages 46–51, May 2004.

[103] IEEE standard test access port and boundary scan architecture. *IEEE Standard 1149.1-2001*, pages 1–212, July 2001.

[104] B. Vermeulen and S. K. Goel. Design for debug: catching design errors in digital chips. *IEEE Design Test of Computers*, 19(3):35–43, May 2002.

[105] D. D. Josephson. The manic depression of microprocessor debug. In *IEEE International Test Conference (ITC)*, pages 657–663, 2002.

[106] D. Josephson and B. Gottlieb. The crazy mixed up world of silicon debug [IC validation]. In *IEEE Custom Integrated Circuits Conference*, pages 665–670, Oct 2004.

[107] H. F. Ko and N. Nicolici. Automated trace signals selection using the RTL descriptions. In *IEEE International Test Conference (ITC)*, pages 1–10, Nov 2010.

[108] C. MacNamee and D. Heffernan. Emerging on-ship debugging techniques for real-time embedded systems. *IEEE Journal of Computing Control Engineering*, 11(6):295–303, Dec 2000.

[109] C. B. Stunkel, B. Janssens, and W. K. Fuchs. Address tracing for parallel machines. *IEEE Computers*, 24(1):31–38, Jan 1991.

[110] E. E. Johnson, Jiheng Ha, and M. Baqar Zaidi. Lossless trace compression. *IEEE Transactions on Computers (TCOMP)*, 50(2):158–173, Feb 2001.

[111] A. B. T. Hopkins and K. D. McDonald-Maier. Debug support strategy for systems-on-chips with multiple processor cores. *IEEE Transactions on Computers (TCOMP)*, 55(2):174–184, Feb 2006.

[112] L.B. Arimilli, M.S. Floyd, L.S. Leitner, K.F. Reick, and J.L. Vargus. Multistate logic analyzer integral to a microprocessor, October 14 2003. US Patent 6,633,838.

[113] A.L. Herrmann and G.P. Nugent. Embedded logic analyzer for a programmable logic device, January 30 2001. US Patent 6,182,247.

[114] Xilinx Verification Tool. ChipScope. `http://www.xilinx.com`. 2006.

[115] Altera Verification Tool. SignalTap II Embedded Logic Analyzer. `http://www.altera.com`. 2006.

[116] Yu-Chin Hsu, Furshing Tsai, Wells Jong, and Ying-Tsai Chang. Visibility enhancement for silicon debug. In *ACM/IEEE Design Automation Conference (DAC)*, pages 13–18, 2006.

[117] R. Leatherman and N. Stollon. An embedding debugging architecture for socs. *IEEE Potentials*, 24(1):12–16, Feb 2005.

[118] H. F. Ko and N. Nicolici. Mapping trigger conditions onto trigger units during post-silicon validation and debugging. *IEEE Transactions on Computers (TCOMP)*, 61(11):1563–1575, Nov 2012.

[119] D. Lin, E. Singh, C. Barrett, and S. Mitra. A structured approach to post-silicon validation and debug using symbolic quick error detection. In *IEEE International Test Conference (ITC)*, pages 1–10, Oct 2015.

[120] D. Lin, S. Eswaran, S. Kumar, E. Rentschler, and S. Mitra. Quick error detection tests with fast runtimes for effective post-silicon validation and debug. In *ACM/IEEE Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1168–1173, March 2015.

[121] 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 (TCAD)*, 33(10):1573–1590, Oct 2014.

[122] D. Lin, T. Hong, Y. Li, F. Fallah, D. S. Gardner, N. Hakim, and S. Mitra. Overcoming post-silicon validation challenges through quick error detection (QED). In *ACM/IEEE Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 320–325, March 2013.

[123] N. R. Saxena, S. Fernandez-Gomez, Wei-Je Huang, S. Mitra, Shu-Yi Yu, and E. J. McCluskey. Dependable computing and online testing in adaptive and configurable systems. *IEEE Design Test of Computers*, 17(1):29–41, Jan 2000.

[124] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers (TCOMP)*, 37(2):160–174, Feb 1988.

[125] D. J. Lu. Watchdog processors and structural integrity checking. *IEEE Transactions on Computers (TCOMP)*, C-31(7):681–685, July 1982.

[126] C. Wang, H. s. Kim, Y. Wu, and V. Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *International Symposium on Code Generation and Optimization (CGO)*, pages 244–258, March 2007.

[127] N. R. Saxena and E. J. McCluskey. Dependable adaptive computing systems-the ROAR project. In *IEEE International Conference on Systems, Man, and Cybernetics*, volume 3, pages 2172–2177 vol.3, Oct 1998.

[128] B. Vermeulen, M. Z. Urfianto, and S. K. Goel. Automatic generation of breakpoint hardware for silicon debug. In *ACM/IEEE Design Automation Conference (DAC)*, pages 514–517, July 2004.

[129] 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)*, pages 982–985, May 2016.

[130] R. Sebastian, S. R. Mary, Gayathri 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)*, pages 585–588, Nov 2015.

[131] H. Sohofi and Z. Navabi. Assertion-based verification for system-level designs. In *IEEE International Symposium on Quality Electronic Design (ISQED)*, pages 582–588, March 2014.

[132] Yael Abarbanel, Ilan Beer, Leonid Gluhovsky, Sharon Keidar, and Yaron Wolfsthal. *FoCs – Automatic Generation of Simulation Checkers from Formal Specifications*, pages 538–542. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

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

[134] M. Boulé and Z. Zilic. *Generating Hardware Assertion Checkers: For Hardware*

Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring. Springer, 2008.

[135] M. Boule and Z. Zilic. Efficient automata-based assertion-checker synthesis of PSL properties. In *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pages 69–76, Nov 2006.

[136] M. R. Kakoee, M. H. Neishaburi, M. Daneshtalab, S. Safari, and Z. Navabi. On-chip verification of NoCs using assertion processors. In *Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD)*, pages 535–538, Aug 2007.

[137] F. C. Sica, C. N. Coelho, J. A. M. Nacif, H. Foster, and A. O. Fernandes. Exception handling in microprocessors using assertion libraries. In *Symposium on Integrated Circuits and Systems Design*, pages 55–59, Sept 2004.

[138] K. Peterson and Y. Savaria. Assertion-based on-line verification and debug environment for complex hardware systems. In *IEEE International Symposium on Circuits and Systems*, volume 2, pages II–685–8 Vol.2, May 2004.

[139] M. H. Neishaburi and Z. Zilic. On a new mechanism of trigger generation for post-silicon debugging. *IEEE Transactions on Computers (TCOMP)*, 63(9):2330–2342, Sept 2014.

[140] J.H. Barton, E.W. Czeck, Z.Z. Segall, and D.P. Siewiorek. Fault injection experiments using FIAT. *IEEE Transactions on Computers (TCOMP)*, 39(4):575–582, Apr 1990.

[141] D. Chatterjee, C. McCarter, and V. Bertacco. Simulation-based signal selection for state restoration in silicon debug. In *ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 595–601, Nov 2011.

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

[143] Yu-Shen Yang, N. Nicolici, and A. Veneris. Automated data analysis solutions to silicon debug. In *ACM/IEEE Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 982–987, April 2009.

[144] A. Hekmatpour and A. Salehi. Block-based schema-driven assertion generation for functional verification. In *IEEE Asian Test Symposium (ATS)*, pages 34–39, Dec 2005.

[145] G. Pinter and I. Majzik. Automatic generation of executable assertions for runtime checking temporal requirements. In *IEEE International Symposium on High-Assurance Systems Engineering*, pages 111–120, Oct 2005.

[146] Po-Hsien Chang and L.-C. Wang. Automatic assertion extraction via sequential data mining of simulation traces. In *(ACM/IEEE) Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 607–612, Jan 2010.

[147] O. Kupferman, Wenchao Li, and S.A. Seshia. A theory of mutations with applications to vacuity, coverage, and fault tolerance. In *Formal Methods in Computer-Aided Design*, pages 1–9, Nov 2008.

[148] S. Das, A. Banerjee, P. Basu, P. Dasgupta, P.P. Chakrabarti, C.R. Mohan, and L. Fix. Formal methods for analyzing the completeness of an assertion suite against a high-level fault model. In *IEEE International Conference on VLSI Design*, pages 201–206, Jan 2005.

[149] Shuo Sheng and M.S. Hsiao. Efficient sequential test generation based on logic simulation. *IEEE Design Test of Computers*, 19(5):56–64, Sep 2002.

[150] A Parikh, Weixin Wu, and M.S. Hsiao. Mining-guided state justification with partitioned navigation tracks. In *IEEE International Test Conference (ITC)*, pages 1–10, Oct 2007.

[151] M. Dusanapudi, S. Fields, M.S. Floyd, G.L. Guthrie, R. Kalla, S. Kapoor, L.S. Leitner, C.F. Marino, J.J. McGill, A. Nahir, K. Reick, H. Shen, and K.L. Wright. Debugging post-silicon fails in the IBM POWER8 bring-up lab. *IBM Journal of Research and Development*, 59(1):12:1–12:10, Jan 2015.

[152] M. Gao, P. Lisherness, and K. T. Cheng. Post-silicon bug detection for variation induced electrical bugs. In *ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 273–278, Jan 2011.

[153] A. Margulis, D. Akselrod, E. Rentschler, and M. Ricchetti. Evolution of graphics northbridge test and debug architectures across four generations of AMD ASICs. *IEEE Design Test*, 30(4):16–25, Aug 2013.

[154] B. Bentley. Validating the Intel Pentium 4 Microprocessor. In *ACM/IEEE Design Automation Conference (DAC)*, pages 244–248, 2001.

[155] G. Stoler. Validation of complex designs through hardware prototyping. In *IEEE Computer Society Workshop on VLSI*, pages 149–154, 2000.

[156] Chung-Yang Huang, Yu-Fan Yin, Chih-Jen Hsu, T.B. Huang, and Ting-Mao Chang. SoC HW/SW verification and validation. In *ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 297–300, Jan 2011.

[157] K. Balston, A. J. Hu, S. J. E. Wilton, and A. Nahir. Emulation in post-silicon validation: It's not just for functionality anymore. In *IEEE High Level Design Validation and Test Workshop (HLDVT)*, pages 110–117, Nov 2012.

[158] R. O. Gallardo, A. J. Huy, A. Ivanov, and M. S. Mirian. Reducing post-silicon coverage monitoring overhead with emulation and bayesian feature selection. In *ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 816–823, Nov 2015.

[159] Ben Cohen. *Using PSL/Sugar with Verilog and VHDL, Guide to Property Specification Language for ABV*. VhdlCohen Publishing, 5 2003.

[160] Srikanth Vijayaraghavan and Meyyappan Ramanathan. *A Practical Guide for SystemVerilog Assertions*. Springer, 2005 edition, 6 2005.

[161] S. Das, R. Mohanty, P. Dasgupta, and P. P. Chakrabarti. Synthesis of system verilog assertions. In *Proceedings of the ACM/IEEE Design Automation Test in Europe Conference (DATE)*, volume 2, pages 1–6, March 2006.

[162] C. Fibich, M. Wenzl, and P. Rssler. On automated generation of checker units

from hardware assertion languages. In *Microelectronic Systems Symposium (MESS), 2014*, pages 1–6, May 2014.

[163] M. Wenzl, C. Fibich, P. Rssler, H. Taucher, and M. Matschnig. Logic synthesis of assertions for saftey-critical applications. In *IEEE International Conference on Industrial Technology (ICIT)*, pages 1581–1586, March 2015.

[164] Kwang-Ting Cheng, Shi-Yu Huang, and Wei-Jin Dai. Fault emulation: A new methodology for fault grading. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 18(10):1487–1495, Oct 1999.

[165] C. Lopez-Ongil, M. Garcia-Valderas, M. Portela-Garcia, and L. Entrena-Arrontes. Techniques for fast transient fault grading based on autonomous emulation. In *ACM/IEEE Design, Automation and Test in Europe (DATE)*, pages 308–309 Vol. 1, March 2005.

[166] IEEE standard for SystemVerilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009)*, pages 1–1315, Feb 2013.

[167] Altera Corporation. `http://www.altera.com`. 2015.

[168] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[169] J. P. Marques Silva and K. A. Sakallah. GRASP-A new search algorithm for satisfiability. In *ACM/IEEE International Conference on Computer-Aided Design*, pages 220–227, Nov 1996.

[170] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *ACM/IEEE Design Automation Conference*, pages 530–535, June 2001.

[171] Y. Vizel, G. Weissenbacher, and S. Malik. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE*, 103(11):2021–2035, Nov 2015.

[172] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 21(12):1377–1394, Dec 2002.

[173] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *ACM/IEEE Design Automation Conference (DAC)*, pages 317–320, 1999.

[174] Y. Hu, V. Shih, R. Majumdar, and L. He. Exploiting symmetries to speed up sat-based boolean matching for logic synthesis of fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(10):1751–1760, Oct 2008.

[175] S. Sapra, M. Theobald, and E. Clarke. SAT-based algorithms for logic minimization. In *IEEE International Conference on Computer Design (ICCD)*, pages 510–517, Oct 2003.

[176] V. Khomenko, M. Koutny, and A. Yakovlev. Logic synthesis for asynchronous

circuits based on Petri net unfoldings and incremental SAT. In *IEEE International Conference on Application of Concurrency to System Design*, pages 16–25, June 2004.

[177] Yung-Chieh Lin, Feng Lu, Kai Yang, and Kwang-Ting Cheng. Constraint extraction for pseudo-functional scan-based delay testing. In *ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, volume 1, pages 166–171 Vol. 1, Jan 2005.

[178] R. Drechsler, S. Eggergluss, G. Fey, A. Glowatz, F. Hapke, J. Schloeffel, and D. Tille. On acceleration of sat-based atpg for industrial designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1329–1333, July 2008.

[179] P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 15(9):1167–1176, Sep 1996.

[180] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *ACM Communications*, 5(7):394–397, July 1962.

[181] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers (TCOMP)*, 35(8):677–691, August 1986.

[182] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *National Conference on Artificial Intelligence, AAAI*, pages 440–446. AAAI Press, 1992.

[183] Mary Sheeran and Gunnar Stålmarck. A tutorial on stålmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16(1):23–58, 2000.

[184] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

[185] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. IJCAI, pages 399–404, 2009.

[186] ABC: A system for sequential synthesis and verification. `https://people.eecs.berkeley.edu/~alanmi/abc/`.

[187] IEEE standard for property specification language (PSL). *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pages 1–182, April 2010.

[188] 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)*, pages 148–155, March 2014.

[189] A. Vali and N. Nicolici. Bit-flip detection-driven selection of trace signals. In *IEEE European Test Symposium (ETS)*, pages 1–6, May 2016.

[190] X. Shi and N. Nicolici. On-chip constrained random stimuli generation for post-silicon validation using compact masks. In *IEEE International Test Conference (ITC)*, pages 1–10, Oct 2014.