A TEST VECTOR MINIMIZATION ALGORITHM BASED ON DELTA DEBUGGING FOR POST-SILICON VALIDATION OF PCIe ROOTPORT

By

TOH YI FENG

A Dissertation submitted for partial fulfilment of the requirement

for the degree of Master of Science

(Electronic Systems Design Engineering)

August 2017

ACKNOWLEDGEMENT

I would like to take this opportunity to thank all people who have been very supportive throughout my thesis writing. I have to thank my loving parents for providing moral support, as thesis writing is a tough journey, especially while working full-time job at the same time.

I have to thank my thesis supervisor, Dr Teh, for providing valuable advice and comments on thesis writing, and pointing a clear direction on doing better research and writing a better thesis.

I am also grateful to have a caring and understanding manager at work, Mr. Heng, who supported me to pursue a Master programme despite having to work full-time. Without his help, I wouldn't have the resources to evaluate the performance of the minimizer algorithm.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

BDD	Binary Decision Diagram
BE	Byte Enable (used in Last DW BE and 1st DW BE in TLP field)
BFM	Bus Functional Model
BIOS	Basic Input/Output System
BMC	Bounded Model Checking
CPU	Central Processing Unit
DW	Double Word (Data size convention in Intel 32bits or 4Bytes)
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IC	Integrated Circuit
IP	Intellectual Property (usually referring to silicon design in the form of HDL)
LFSR	Linear Feedback Shift Register
MTRR	Memory Type Range Register
OBDD	Ordered Binary Decision Diagrams
PCH	Platform Controller Hub
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
RAM	Random Access Memory
RTL	Register-Transfer Level (a category of HDL code)
SAT	Boolean Satisfiability
SoC	System-on-Chip
SUT	System-under-test
TLP	Transaction Layer Packet
UC	Uncachable memory type
WB	Write-Back (memory cache type)
WC	Write-Combining (memory cache type)
WP	Write-Protect (memory cache type)
WT	Write-Through (memory cache type)

Algorithma Minimisasi Vektor Ujian Bedasarkan Delta Debugging untuk Pengesahan Pasca-Silikon dalam Rekabentuk PCIe Rootport

ABSTRAK

Dalam reka bentuk peranti silikon, contohnya peranti PCIe, pengesahan reka bentuk adalah bahagian yang penting dalam proses reka bentuk, di mana peranti tersebut diuji dengan ujian pengesahan yang mengesahkan fungsinya. Walau bagaimanapun, penyahpepijatan(debugging) manual masih digunakan secara meluas dalam pengesahan pasca-silikon dan sebilangan besar ujian vektor perlu dianalisis dan ianya melambatkan proses. Justeru, suatu algoritma minimisasi vektor ujian telah dicadangkan untuk menghapuskan vektor ujian berlebihan yang tidak menyumbang kepada penghasilan semula(reproduction) kegagalan ujian. Kaedah yang dicadangkan diilhamkan oleh algoritma "Delta Debugging" yang digunakan dalam penyahpepijatan automatik perisian tetapi masih belum diaplikasikan dalam penyahpepijatan pasca-silikon. Kaedah ini beroperasi dengan menggunakan prinsip pembahagian binari vektor ujian secara binary, dan menguji setiap subset pada satu sistem pasca-silikon "Sistem-Under-Test" (SUT) untuk menentukan jika subset boleh dihapuskan. Apabila diuji dengan menggunakan set ujian vektor yang mengandungi ujian vektor salah dengan sengaja, algoritma ini dapat mengeluarkan vektor ujian salah dengan baik. Dalam kes-kes ujian yang mengandungi sehingga 10,000 vektor ujian, minimizer hanya mengambil masa kira-kira 16ns untuk setiap ujian vektor dalam kes ujian apabila ianya hanya mengandungi satu vektor ujian yang salah. Dalam kes ujian dengan 1000 vektor termasuk vektor yang salah, ia mengambil masa kira-kira 140μs setiap ujian vektor salah yang disuntik. Dengan itu penggunaan CPU minimizer sangat kecil jika dibandingkan dengan masa ujian yang dijalankan pada SUT. Faktor-faktor yang memberi impak besar kepada prestasi algoritma, adalah bilangan vektor salah dan penaburan (jarak) vektor salah. Kesan daripada jumlah vektor ujian dan kedudukan vektor salah agak kecil berbanding dengan dua yang lain. Oleh itu, algoritma ini paling berkesan bagi kes-kes di mana terdapat hanya beberapa vektor ujian yang salah, dengan set vektor ujian yang besar.

A Test Vector Minimization Algorithm Based on Delta Debugging For Post-Silicon Validation of PCIe Rootport

ABSTRACT

In silicon hardware design, such as designing PCIe devices, design verification is an essential part of the design process, whereby the devices are subjected to a series of tests that verify the functionality. However, manual debugging is still widely used in post-silicon validation and is a major bottleneck in the validation process. The reason is a large number of tests vectors have to be analyzed, and this slows process down. To solve the problem, a test vector minimizer algorithm is proposed to eliminate redundant test vectors that do not contribute to reproduction of a test failure, hence, improving the debug throughput. The proposed methodology is inspired by the Delta Debugging algorithm which is has been used in automated software debugging but not in post-silicon hardware debugging. The minimizer operates on the principle of binary partitioning of the test vectors, and iteratively testing each subset (or complement of set) on a post-silicon System-Under-Test (SUT), to identify and eliminate redundant test vectors. Test results using test vector sets containing deliberately introduced erroneous test vectors show that the minimizer is able to isolate the erroneous test vectors. In test cases containing up to 10,000 test vectors, the minimizer requires about 16ns per test vector in the test case when only one erroneous test vector is present. In a test case with 1000 vectors including erroneous vectors, the same minimizer requires about 140µs per erroneous test vector that is injected. Thus, the minimizer's CPU consumption is significantly smaller than the typical amount of time of a test running on SUT. The factors that significantly impact the performance of the algorithm are number of erroneous test vectors and distribution (spacing) of the erroneous vectors. The effect of total number of test vectors and position of the erroneous vectors are relatively minor compared to the other two. The minimization algorithm therefore was most effective for cases where there are only a few erroneous test vectors, with large number of test vectors in the set.

CHAPTER 1. INTRODUCTION

1.1 Background

In the electronics industry, one of the currently most prominent subdomain is microelectronics. As the name implies, microelectronics involves the design and manufacturing of electronic devices based on extremely tiny structures in semiconductor. Almost all digital electronic devices nowadays take advantage of the microelectronics to manufacture highly complex circuits on silicon, containing millions or billions of transistors on a single silicon die, which is then packaged into an Integrated Circuit (IC). These products are so common, and to put things into perspective, microprocessors and System-on-Chip (SoC) in our computers and smartphones are examples of IC.

The sheer complexity of IC introduces high chances of design errors or bugs. When it is compared to simpler devices, a complex device has more possible points of failure. For this reason, Hardware Verification and Validation (V&V) or "verification" for short, has been an important activity in developing such complex IC. Verification can be described as a series of processes that include testing the device, locating bugs and solving bugs to ensure that a device operates properly according to its specifications ("IEEE Draft Standard for System, Software and Hardware Verification and Validation," 2015). Failure to ensure bug-free ICs before delivering them to customer has far-reaching effects. For example, if such buggy ICs are used in healthcare devices or industrial robots, they will lead to unreliable operation or even catastrophic harm to the device or to the users. Buggy devices will also impose a significant financial penalty to the company, for example, the Pentium FDIV bug costed Intel at least 475 million USD in order to replace all faulty devices (Pratt, 1995).

Verification activities can be characterized in many ways. In terms of scope, verification of a digital IC ranges from as small as an IP block (or module) to full-chip level and finally to

system-level, as shown in Figure 1.1. The IP block verification tests an IC module, for example network controller within a smartphone SoC. Full-chip verification tests the whole chip at once. System-level verification tests the IC together with the application circuit (for example testing a microprocessor on a motherboard).

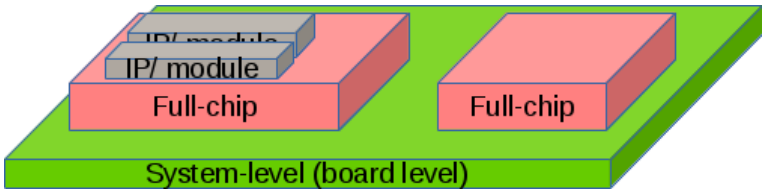

Figure 1.1 The different scopes of verification.

Verification of digital ICs can also be classified according to the stages of product development, roughly into 3 categories such as pre-silicon verification, manufacturing Testing, and Post-silicon verification, as shown in the Figure 1.2. Pre-silicon verification refers to design verification prior to silicon fabrication. Manufacturing testing refers to basic verification during manufacturing of the devices. Post-silicon verification refers to the verification after manufacturing, in final form (silicon die). Generally, pre-silicon verification uses model checking or simulation techniques to verify the correctness of the design. Manufacturing testing uses consistency checks to verify that all devices behave the same as a way to rule out fabrication defects. Post-silicon verification typically runs verification workloads on real environments (such as on a consumer product), and it is used to detect and address issues that only manifest in real silicon (Mitra et al., 2010).

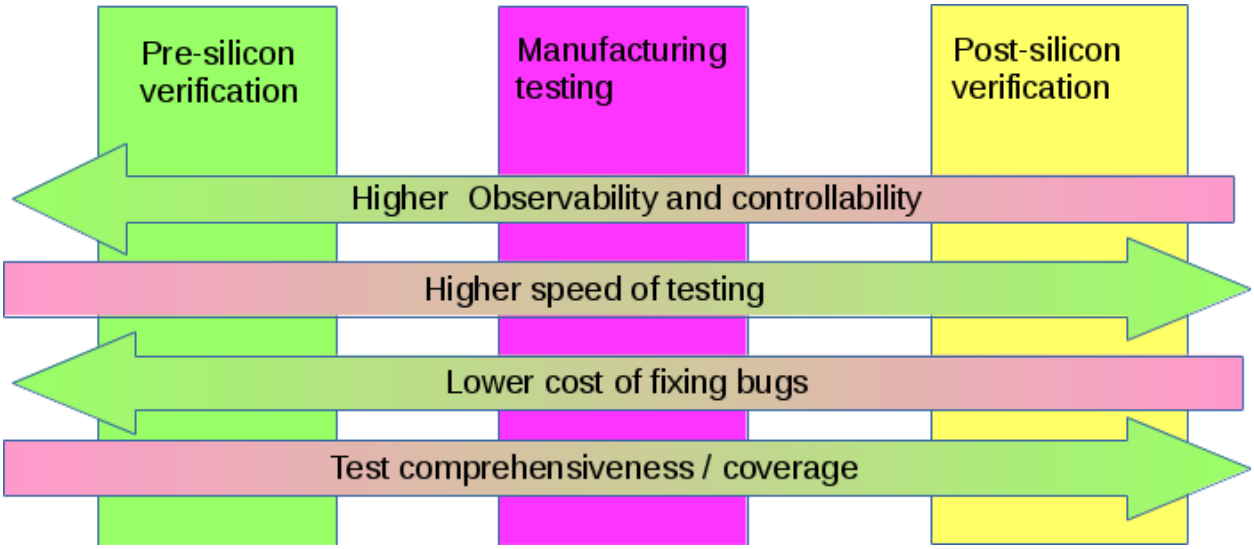

Figure 1.2 Diagram depicting the spectrum of characteristics of pre-silicon verification, manufacturing testing and post-silicon verification.

Post-silicon is relatively less standardized than pre-silicon verification as there are no generic methods that can fully verify full design like the formal methods available in pre-silicon verification. There are, however, ongoing researches that try to bridge pre-silicon methods with post-silicon methods, and thus, post-silicon verification is an emerging field of research in the sphere of hardware V&V. Even though pre-silicon and post-silicon verification share similar traits, they are sufficiently different to make both of them equally important. Most notably, post-silicon verification has a much higher speed than pre-silicon methods (pre-silicon methods can't run at real-time speed like real device does), while pre-silicon methods has better debuggability and observability (Wagner and Bertacco, 2011). Researches are being done to improve debuggability of post-silicon verification methods.

PCIe (Peripheral Component Interconnect Express) is a high speed serial bus that is used for connecting a wide array of peripheral devices, such as graphics cards, networking cards, storage devices and etc. to a computer system. It is a replacement to the PCI and PCI-X buses that are based on a shared parallel bus that is bandwidth scalability limited. Unlike the predecessor PCI bus,

PCIe is a point-to-point link, which uses separate set of receive and transmit differential pair wires to connect devices. It is packet-based protocol, with a 3-layered protocol in the hardware to handle the packets which are Transaction Layer, Data Link Layer and Physical Layer. A data transaction such as “memory write”, is turned into a data packet in Transaction Layer, and pass through subsequently lower layers, the Data Link Layer and Physical Layer, for further processing, and it is sent through the PCIe link to the device on the other end of the link, where the data packet will traverse the layers in reverse, and arrives in Transaction layer where it will finally be decoded into a memory write transaction, as shown in Figure 1.3.

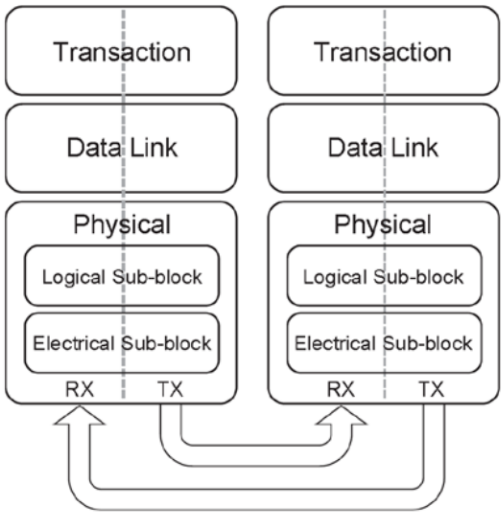

Figure 1.3 Logical layers of PCI Express (showing two devices on a single link).

PCIe protocol is robust, flexible, scalable and highly configurable. For example, a PCIe link can support a link width configuration ranging from x1 to x32 with a step increase of power-of-two (a link width of x1 refers to 1-lane link, which consists of a transmit and a receive differential pairs per lane) depending on the maximum link width shared by the devices on the two ends of a link. Link speed is also configurable, from 2.5Gbps to 8Gbps per lane, depending on the required bandwidth or power savings requirement. The PCIe bus is not just used for transferring data but it also has the capability to handle power management events, hot plug events, interrupts, access

control, local time synchronization across devices, traffic prioritization and etc. All of these require complex circuits to support the features. Thus, PCIe is both a communication link but also virtually forms the backbone of a computer system.

A PCIe device can be either a root port, an endpoint, a switch or a bridge. The simplest PCIe system consists of a root port connected to an endpoint with a PCIe link. Another example of PCIe system is shown in Figure 1.4. The root port refers to the PCIe port that is exposed by the host system, typically a computer chipset or a microprocessor, while an endpoint refers to a device (for example, graphics card) that connects to the rootport. This dissertation will specifically address the verification of a PCIe rootport of an Intel PCH. The PCIe rootport is considered as an IP block by itself, so verification of a PCIe rootport is considered as IP-level verification.

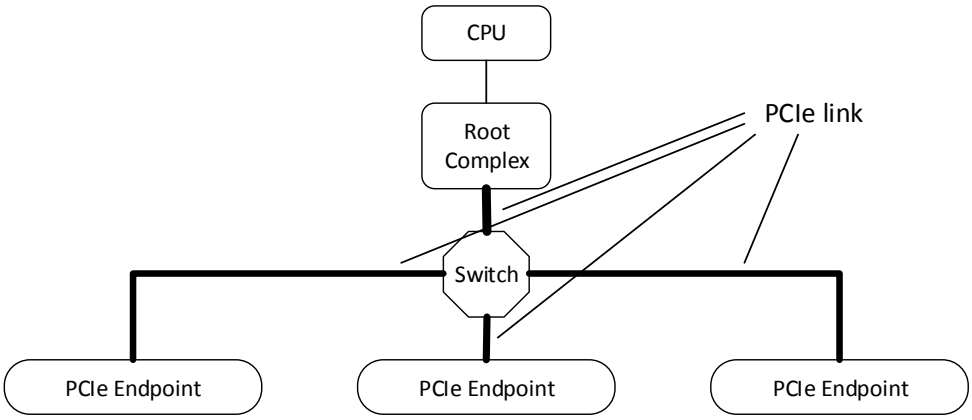

Figure 1.4 An example hierarchy of a PCIe system, consisting of a root complex (hosting a root port), a switch and 3 endpoints.

The complexity of the PCIe bus protocol is daunting. The compliance checklist for an endpoint exceeds 1000 items (Endpoint Compliance Checklist for the PCI Express Base 1.0a Specification, 2004). Couple this with the number of configurable features, the permutations for all possible verification test cases may exceed millions. In short, it has an extremely large verification space. Moreover, multiple clock boundaries, and massive state machines (often containing sub-state machines in each state) and long simulation time to simulate certain features and state

transitions also further increase the complexity (Emara et al.,2005). The PCIe specification separates the PCIe functionality into 3 distinct layers, e.g. the transaction layer, the data link layer and the physical layer, as shown in Figure 1.3. The purpose of this separation is to ease the work of designing a PCIe device as well as reducing the difficulty of verification. Despite this, the verification of PCIe root port IP is still a challenging task for both pre and post-silicon. Post silicon validation is especially important as the PCIe subsystem closely interacts with the whole computer system in many aspects, for instance, power management (e.g. turn on, sleep, standby etc.). Moreover, physical effects such as random link noise and data corruptions (which the PCIe IP must handle gracefully) can only be tested in post-silicon. Therefore, a PCIe rootport has to be verified in post-silicon together with the systems surrounding it (such as memory controllers, CPU etc.) under a condition that mimics a realistic workload.

Two main approaches of post-silicon verification of PCIe rootport are the directed focus tests and random concurrency tests. The former is a more focused approach that only exercise one feature to maximize the number of iterations for a specific feature while the latter involves running tests on all features at the same time to maximize the interactions between different features. Random concurrency tests has become the major test for PCIe, especially for regression tests, as it is able to uncover more corner-case bugs that are not usually detected in focus tests. It is worth highlighting that most of the random concurrency tests are in fact pseudorandom. This means that the test sequences appear to be random but are predictable and reproducible in order to aid in bug reproduction and debugging.

A random concurrency test usually has three major phases, which are setup phase, execute phase and verify phase, as shown in Figure 1.5. Setup phase initializes the hardware in preparation for the test, execute phase runs the tests (e.g. sends the data up and down the link without verifying them), and verify phase checks for any errors or coherency issues. The stimuli that drive the

hardware during execution of a test are referred to as test vectors, with each test vector being a discrete action that will generate a result.

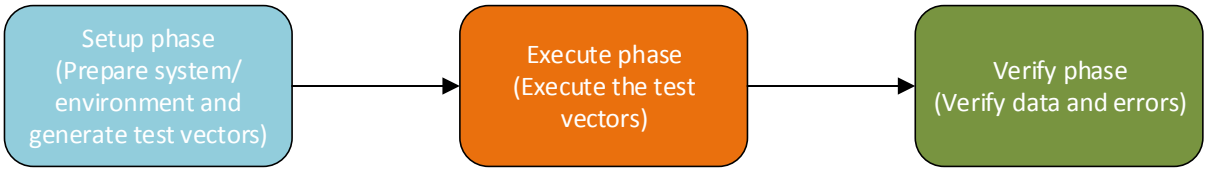

Figure 1.5 The 3 main phases of a typical random concurrency test.

Random concurrency tests are used most often for testing PCIe rootport as this scheme maximizes the stressing of PCIe bus in the execution phase, allowing more corner cases to be covered. Stressing is achieved by making sure that verification activity which causes bus idling doesn't occur during test execution phase.

1.2 Problem Statement

An inherent drawback of the random concurrency test approach is the long error detection latency. This means that when an error occurs during the execute phase, it may not be detected immediately until it enters verification phase. When verification failure occurs, the test vectors that lead to the error have to be manually determined. However, since the random stimuli concurrency tests involve a large set of test vectors per unique iteration, it is very difficult to manually examine them all in a reasonable amount of time. Often, such ICs provides a means of capturing some of the internal signals. Otherwise, a logic analyzer can be used to capture internal signals of the IC, if it is deliberately exposed. However, there is a limit on the number of signals and the amount of signal data that can be captured. This is due to the inability to fully capture the signals for the execution of the full set of vectors that can take more than a million clock cycles.

In most cases, only some of the test vectors in a test case are responsible for the error reproduction. As the slow manual debugging process is a major bottleneck in post-silicon validation

effort, it is vital to reduce the amount of the input test vectors using an automated approach due to tight marketing schedule.

1.3 Objectives

1. To enhance a PCIe post-silicon random concurrency test by implementing an algorithm that automatically minimize the number of test vectors for reproducing bug.
    - Reducing the size of test vectors by eliminating non-error-causing test vectors improves debugging efficiency as less time is wasted on identifying the real cause of an issue.
2. To determine the execution speed of the proposed test vector minimization algorithm.
    - The speed of minimizer algorithm is measured by determining the total number of tests executed on the SUT by the minimizer program, and CPU time spent within the minimizer algorithm.
3. To determine the efficiency of the proposed test vector minimization algorithm when subjected to different scenarios.
    - The minimizer algorithm is subjected to different input parameters such as size of a set of test vector in each test case (each test case has exactly one set of test vectors), number of error-inducing test vectors, position of error-inducing test vectors and distribution of error-inducing test vectors. The speed of execution under the different conditions are compared.

1.4 Scope

The focus of this work is to improve the verification software used for testing PCIe root port in post silicon verification. This thesis focuses on the study and the development of a software algorithm that is used in conjunction with an existing random-concurrency test for post-silicon verification. The aim is to enhance the existing test software and test content without introducing additional hardwares.

This research does not cover:

(i) Hardware enhancement and methods to aid debuggability of post-silicon verification

(ii) Model-based verification (existing random-concurrency test do not require the mathematical model of the PCIe root port to be known)

(iii) Verification of other digital circuits other than PCIe such as microprocessors and other subsystems

1.5 Thesis Outline

The thesis begins with Chapter 1 that provides overview of the silicon design validation methodologies, the PCIe technology and the motivation behind this research. Chapter 2 compares and contrasts in detail the current state of the art of hardware verification, software verification methodologies and debugging methodologies, as well as PCIe-specific verification methodologies. Chapter 3 outlines the implementation of test vector minimization algorithm and also steps to evaluate the algorithm's efficiency. The performance data of the minimizer algorithm is discussed in Chapter 4. The dissertation concludes in Chapter 5 with additional comments on future improvements.

CHAPTER 2. LITERATURE REVIEW

2.1 Overview

This section discuss about the current state of the art of testing/verification/validation methodologies that are currently being used in various fields. The section begins with the discussion of hardware verification methodologies, followed by software testing/verification methodologies, and finally PCIe-specific testing methodologies. A summary of the findings is given and the available gaps are identified.

2.2 Hardware Verification

Hardware verification usually refers to the functional verification of a complex digital device such as microprocessor. The terminology may also differ as some manufacturers and even some research papers refer to it as "Validation". Hardware functional verification is part of the design cycle and it commences immediately after a digital circuit is designed with Hardware Description Language (HDL) code such as Verilog and VHDL. If a design error is found during verification stage, the HDL code will be modified, and then verifications are repeated.

Hardware verification is a challenging task due to the enormous number of possible scenarios (in terms of inputs and current state of state machines). For example, a simple design with 100 flip-flops has a total of $2^{100} \approx 10^{30}$ combinations of possible states, and for a moderately complex design with 512 flip-flops has about $2^{512} \approx 10^{154}$ combinations which are impossible to be fully verified in a lifetime. Therefore, the number of possible test scenarios increases exponentially with the number of transistor in the design. Hence the industry typically spends more than 50% of their effort in terms of cost and time in verification activities during the design phase (Narayanasamy et al., 2006). Nonetheless, errors continue to slip through to customers, and it is common for manufacturers to release documents known as errata sheets that describe device bugs and workarounds.

All hardware verifications aim to find and fix issues in the design. However, there is no single perfect verification algorithm or methodology that works for all applications and scenarios. Each of the verification methodologies has its own set of strength and weaknesses. Therefore, it is necessary to review the currently available ways for hardware verification.

2.2.1 Pre-Silicon and Post-Silicon Verification

The two major types of design verification are pre-silicon and post-silicon verification. Pre-silicon verification is carried out directly by processing the design HDL codes to verify that the code are written correctly and function according to the specifications. Post-silicon verification is carried out after the HDL code has been converted into a real device (i.e. manufactured), usually in a hardware and software environment same as the customer's configuration.

Generally, pre-silicon verification benefits from good controllability and observability since the entire design in pre-silicon stage is still in the form of a HDL code that can be observed and modified easily. The HDL code represents the model of the device, thus with the use of an appropriate software algorithm, the behavior of the device can be reproduced and studied easily. Any signals in the circuit can be observed at any time. In contrast, in post-silicon verification, controllability and observability is limited as all the signals are physically embedded in the semiconductor chip, and only a limited number of signals can be broken out using pins at any time. Once a bug is found during pre-silicon, the fix only requires code changes, but a bug fix during post-silicon requires circuit editing or respinning a new mask or bypass the problem by patching firmware, which is very costly. Moreover, it is easier to quantify verification coverage during pre-silicon phase since the design is fully transparent to the designers in the form of codes. Typically, the verification coverage metrics include code coverage, assertion coverage and mutation coverage.

On the other hand, post-silicon verification has a speed advantage, as the device is already in the physical form that can be run at full speed. Pre-silicon verification cannot be as fast as post-silicon since it requires computation power to model the behavior of the device (Park, 2010). In addition, pre-silicon verification doesn't take account of physical effects fully, such as electrical noise, signal crosstalk, thermal effects etc. because it would consume too much computational power and slowing down the verification excessively, thus it is common to just verify at the gate level only. Table 2.1 shows the summary of comparison between the pre-silicon verification and post-silicon verification:

Table 2.1 Comparison between pre-silicon verification and post-silicon verification

Pre-silicon verification	**Post-silicon verification**
Good controllability	Limited controllability
Good observability	Limited observability
Speed of verification is slow (Simulation and formal verification are slow)	Speed of verification is high (Direct execution in silicon is fast)
Does not account for physical effects fully, only a the model in form of HDL is tested (Layout, noise etc.)	Takes account of all real environment effects since it is real silicon device
Coverage metrics exist (e.g. code coverage)	Coverage metrics is still open to debate
Cost of fixing bug is low	Cost of fixing bug is high

2.2.2 Formal Verification methods

Formal verification refers to a family of techniques that mathematically verifies that a design will always behave according to the specification under all valid input values (Wagner, 2011). Typically, to use this method, the design will need to be represented using mathematical equations such as Boolean equations, and the equations will be solved to determine if it obeys the design rules and specifications. Certain formal verification methods are also able to generate counterexamples, which is a description of a way that violates the specification, when design inconsistencies are found. Inherently, formal verification methods are best suited for pre-silicon verification.

Formal verification methods can be roughly partitioned into two groups, which is reachability analysis and deductive methods (Kern and Greenstreet, 1999). Model checkers and equivalent checkers are part of reachability analysis while automated theorem provers are example of deductive methods. Due to the vast scope of verification, the properties that are verified using a specific formal verification method are usually limited and usually one form of design specification is used (either RTL level, gate level, abstract function block, or transistor (switch) level) (Cohn 1989). Binary decision diagrams and SAT solvers are the main mathematical tools for formal verification techniques including model checking and theorem proving.

Binary decision diagrams (BDD) are used extensively to represent the Boolean functions of a design. BDDs are data structures in the form of directed acyclic graphs, i.e. a tree, with each node representing a variable in the Boolean function. Each node has two outgoing paths connecting to the next node, with each path corresponds to the value 0 or 1 for the variable. The final nodes that do not have outgoing paths, are not variables but contain the value (result) of the Boolean function. BDDs are able to represent complex Boolean equations in a compact form by eliminating redundant nodes and edges. The canonical form of BDD is known as Ordered Binary Decision Diagram (OBDD), is a BDD with additional constrains such that on all paths from root node to leaf nodes, all variable appears only once, in the same order. Equivalence checking can be performed easily by comparing the OBDD of two circuits. Two circuits which implement the same functionality will have identical OBDD. OBDD can be used to check the possible states a design can reach given all possible inputs. Figure 2.1 illustrates an example of simplification using the BDD technique.

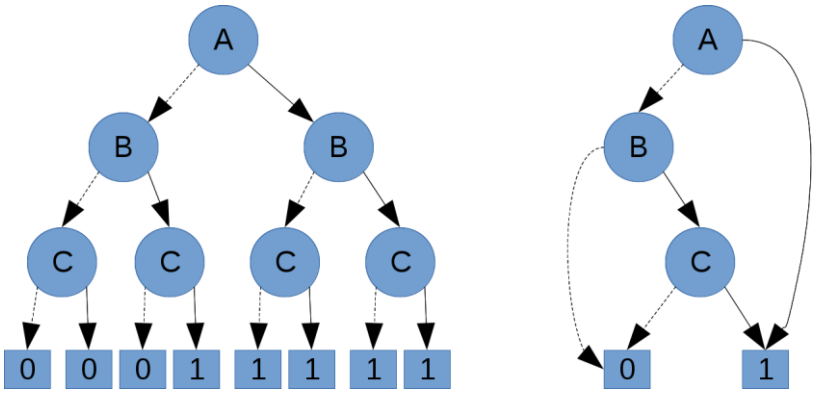

Figure 2.1 Binary Decision Diagrams (BDDs) used for simplification of Boolean equations.

SAT, short for Boolean satisfiability, is a mathematical problem of determining whether there is a set of values, when assigned to the variables of a Boolean function, the function will evaluate to true. The SAT solver is very useful to ensure that a certain erroneous state will not be reached, or to ensure certain signals must follow certain pattern or rules. For example, it can be used to ensure that read and write signals will not be asserted at the same time for a memory interface. A simple SAT algorithm can be implemented using a backtracking search (Gupta et al., 2006). This can be done by first picking an unassigned variable, assign either 0 or 1 to it, and solve the resulting subproblem recursively. At a point where it is determined that there are no solution for the subproblem, the value is flipped and recursed again. If the subproblem can be proved to not satisfy the problem, then there is no need to branch (assign another variable with 0 or 1) further. If there is a set of values that can satisfy the problem, then the SAT problem is said to be satisfiable.

Model checking is a method that verifies that states and transitions in the design exactly follows formal specification. The specifications, or in other words, the desired properties of the design, are specified typically using temporal logic. Temporal logic is used to describe the properties of the system not just in relation to inputs and states, but also time. Usually, as the properties have a specific time window where the formal specification applies, bounded model

checking (BMC) is used, whereby a property is checked only over a limited number of clock cycles and can be deemed sufficient if no violation occurs.

Automated theorem proving is another method to perform formal verification. In this approach, mathematical techniques are used to perform automatic reasoning to formulate the proof for a theorem that is related to the design. The "theorems", as opposed to the Boolean equations used in model checking, are actually abstract (high level) mathematical descriptions of certain properties of the design. It requires human assistance throughout the process of obtaining a proof. Usually, a human will start by proving a simple theorem, then progress to increasingly difficult theorems.

All formal verification methods currently have the same problems. They either require excessively large memory space or excessive long time to finish execution. For example, BDD-based verification may require an extremely large memory to contain all the nodes and edges of the BDD, especially when the BDD can't be optimized significantly due to complexity. SAT is a NP-complete problem, which means it requires an exponential time relative to the input size to complete execution. Tradeoff has to be made between the verification coverage and verification time. Most of the time, the verification coverage or scope has to be very limited in order to allow verification to complete within a reasonable time, unless the design is very simple. The other problem is that the intentions of the designer must be captured precisely in the Boolean equations, models or theorems, and there is no exact approach that can do that flawlessly. Furthermore, some physical effects that only manifest in real silicon can only be determined by experiment, i.e. using post-silicon methods is the only way to test them.

2.2.3 Simulation Methods

Digital logic simulators are also used for verification of IC designs. As the name implies, simulators imitate the real behavior of a device using a pure software approach. Simulation is part

of pre-silicon verification as it doesn't involve hardware implementation of the design. The two basic methods of simulations are compiled simulation and event-driven simulation (Gunes et al., 2005). For design verification, event-driven simulation is always a preferred choice over the other as event-driven simulation takes account of timing in the design more accurately.

Simulation requires the use of simulation models, which is a representation of the design that can be used to determine the behavior of the design under different inputs at different times. Precision of the simulation can be varied depending on the requirement, with low-level simulation being the most precise but comes at a cost of considerably higher computation power and slower since the model will be more complex. From low precision to high precision, the simulation models include behavioral simulation, functional simulation, gate-level simulation, switch-level simulation and transistor-level or circuit level simulation. When different requirements are needed in different parts of the circuit, mixed-mode simulation can be used, whereby several kinds of simulation models can be used to provide more accuracy at certain parts and maintain speed of simulation at the rest.

During simulation, the output of the design-under-test is continuously recorded while being "exercised" by injecting various combination of valid inputs at the same time. To ensure a good coverage, it is often required to inject a large number of unique input values to hit as much scenarios as possible. Therefore, it allows the design to be scrutinized using scenarios that the device will encounter during normal operation, albeit at a much slower rate than the real device.

In simulation, certain specialized models are available for verifying designs involving bus protocols such as PCIe. One example is the Bus Functional Model or also referred to as Transaction Verification Models, allows the simulation to be done to visualize the behavior at a higher level, in terms of bus transactions. For PCIe it is commonly used as simulation test benches to inject bus transactions into a PCIe device-under-test and which then allows responses to be captured and analyzed.

Simulation, especially with precise models like gate-level or finer models, is very slow, because the software has to evaluate the next state of the system by calculating the output states of every logic gates in the design, one by one. Simulation can be accelerated using multiple processors to calculate multiple logic gates at a time, but it is still very slow as there are millions of gates in a practical design.

2.2.4 Prototyping Methods

Prototyping refers to hardware implementation of the design. There are mainly two methods of prototyping, which are hardware emulation or live system (post-silicon). Hardware emulation uses a specialized hardware, typically one or more Field Programmable Gate Array (FPGA) chips, to load the design and execute it. In fact, since FPGA consists of programmable logic gates and circuits, it is able to perform as if it were the fabricated device, only it was somewhat slower compared to the real IC which is a custom-made chip. Instead of using software like what is used in simulation, prototyping FPGA actually uses real hardware logic gates to implement the gates in the design, which all run at the real-time in parallel, therefore it is many orders of magnitude (typically 1000 times or more) faster than software simulation methods. This method is still considered pre-silicon since the device is yet to be fabricated. Being a pre-silicon method allows quick changes so any verification issues can be fixed quickly.

The other method, needless to say, is a method whereby a real device is fabricated in its final form and tested on the final platform of the end user. This method is the so-called post-silicon verification where a device is tested after it is being manufactured. The drawback of the method is the extremely long turn-around-time of the manufacturing process of the device, which may take up to months for devices as complex as the latest microprocessors from Intel. If an error is found in this stage, it would incur a significant cost to re-spin a new mask and a very long time to remanufacture again. Unfortunately, there are many bugs that can only be found when running at

the full silicon speed, especially physical and electrical interactions, thus usually one or two extra re-spinning of masks is inevitable.

2.2.5 Test Vector Generation Methods

For certain verification algorithms like simulation, emulation and post-silicon tests, it is required to generate a set of values as inputs, which is known as test pattern or test vector, to the design-under-verification. The quality of the test vector is very crucial, as it influences the test coverage.

Pseudorandom test vector has been one of the most commonly used technique. Generally it uses a linear feedback shift register (LFSR) to generate a sequence of pseudorandom numbers which is fed into a test vector generator to generate test vectors that can be used by the design-under-test. Given enough numbers of test vector, pseudorandom test can usually provide good test coverage, with low cost of test vector generation (Wagner et al., 1987).

In many cases, there are certain rules to be followed when creating the test vector. For a device that communicates with PCIe protocol, the data packets travelling on the PCIe bus follows certain format, and the values within each field of the PCIe data packets are constrained by the values of another field. Thus, the test vectors are not completely random but is constrained such that all the generated values are in the set of valid values (Shi, 2016). Such test vector generator is called constrained random test vector generator.

However, as the size and complexity of circuit design increase, the verification space also grows enormously, requiring a much longer time to execute more pseudorandom tests vectors in order to obtain sufficient coverage. This creates the necessity to algorithmically generate a compact set of test vectors that can efficiently provide a good test coverage, in terms of high fault detection rate, without requiring very long test time.

Many methods exist for generating compact set of test vectors. Static and dynamic compaction methods are available, whereby the static compaction method eliminates redundant vectors after the full set of vectors are generated, and dynamic compaction method involves embedding of compaction algorithm within the test generation algorithm. One of the early examples include the usage of partitioning and reordering to statically compact the test vectors (Hsiao and Chakradhar, 1998). Another method uses a greedy test compaction at initial phase and incremental dynamic test vector compaction at the later phase which enables better coverage for hard-to-detect faults (Jha, 2013).

There is another test vector generation method that generates test vectors in the form of pairs of complementary operations, which allows an easy verification since it doesn't need to compute the every output for each unique input or test vectors, and the only verification is to make sure that the initial states are unchanged at the end of test (Wagner and Bertacco, 2008).However, this method will not be able to detect complete failure that causes system to hang, which causes the final state (after the test) identical to initial state, and this condition can be mistaken as passing the test.

Test vector compaction has also been used specifically in fault diagnosis, whereby the diagnostic test vectors are reduced to shorten the execution time for fault diagnosis, reduce testing resources and cost of debug (Higami et al., 2006). Test vector compaction in fault diagnosis is relatively unexplored as mentioned by (Higami et al., 2006).

At the time of writing all the existing test vector compaction methods seems to be only applicable for pre-silicon testing as it involves simulation or formal verification, whereby the simulation model of the design is available for analysis by the compaction algorithms. Not to mention, the compaction algorithms are computationally intensive as even the problem of estimating the minimum size test-set for single stuck-at fault it is NP-hard, and heuristic methods

must be employed. Current heuristic methods still doesn't yield optimal test vector set to certain extent (Hochbaum, 1996).

2.3 Software Verification and Debugging Methodologies

In the sphere of software engineering, there are similar counterparts to the hardware verification, and it is known as software testing. Several methodologies in software testing and debugging have been identified to be highly applicable in hardware debugging and related to the work of this thesis and thus presented here.

2.3.1 Random Testing

Given that a typical software nowadays contain a large array of features and are complex as well, software programmers often find themselves having not enough input data to test their software. Traditionally programmers themselves have to think of all possible scenarios to write test cases containing all possible inputs. This approach is not only tedious but is also very limited in terms of coverage. As number of possible paths increases exponentially with number of decision points in the program, the possible number of test cases grows exponentially. (Biere and Brummayer, 2008)

Instead of having to manually prepare the data sets, the data is generated randomly by a random test case generator. Of course, a programmer has to put effort to write such program to generate test cases that can be verified against a known correct value. In long term, it is cheap and scalable for large projects. It is also good for small projects with limited number of programmer that performs the testing. When the random test generator includes invalid inputs in its test cases, it is also called fuzz testing, which is a great method to detect security holes in a software.

2.3.2 Cyclic Debugging

Cyclic debugging is a technique whereby a program is repeatedly executed to reproduce and observe the bug. It is an iterative debugging process whereby a software is re-executed, fast-forwarded to time of failure and making hypothesis and then the cycle repeats until a root cause is found. The method focuses on localizing the region of code that is erroneous. Figure 2.2 illustrates the general flow of cyclic debugging.

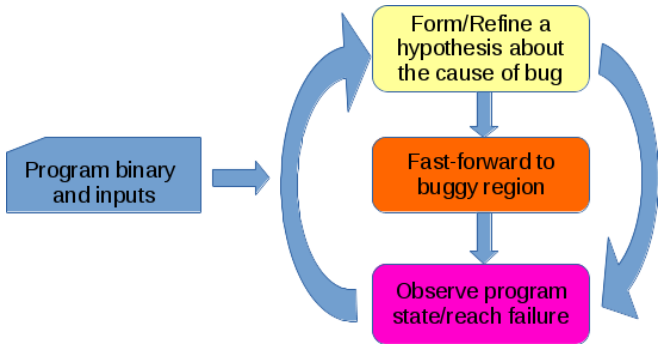

Figure 2.2 Steps involved in Cyclic Debugging.

Cyclic debugging usually involves setting breakpoints in the code at area of interest, executing the program and then observing the state of the program at that breakpoint, to incrementally obtain more information about the failure. However, it can only be used when the program is fully deterministic, such as a single-threaded program or sequential program, which can consistently reproduce the same bug when re-executed repeatedly. In addition, cyclic debugging typically requires manual work to locate the fault, thus it is not very effective for a very large program. However, the repetitive execution pattern has been used as a basis of all other debugging algorithms.

2.3.3 Program Slicing

Program slicing is another debugging method that is based on extraction of a subset of program statements from a given source code, using a specified criteria (Silva 2012). The criteria

is typically based on data dependencies, searching for all program statements that are involved in generating or modifying a particular value. Program slicing greatly eases the fault localization as it reduces the number of codes that a programmer has to analyze for the cause of a bug. By reducing the code size, the interactions of the system that causes the bugs become more apparent to the debugger and hence it can result in quicker and more effective debugging.

Slicing can be done either statically or dynamically. Static slicing only uses the source code itself to provide the clues while dynamic slicing uses information collected during a particular program execution of interest, usually from a program trace. In other words, static slicing extract all codes that modifies the value of a particular variable regardless of whether the codes are executed or not. On the other hand, dynamic slicing determines if a branch (choices in an if-else statement) is taken and only considers the series of codes that are executed during a particular program run. Thus, dynamic slicing is more concise and accurate as compared to static slicing, since it eliminates portions of codes that are not relevant in a failing program execution.

However, program slicing cannot be used alone in software debugging. In most cases, one does not simply know which criteria to be used in performing the program slicing, and a means of obtaining that key information must be sought from other methods.

2.3.4 Delta Debugging

Delta debugging is an automated methodology for debugging programs by finding changes (delta) that causes the failure of a program. Similar to the previous methods, it executes the program repeatedly in a loop to determine failure-inducing changes in the program. The change can be in the form of program input, the program code or even thread schedule. For example, the delta debugging can be used to minimize the set of program code that still produces a failure, by repeatedly remove a small section of program code and re-execute it until a smallest size of program

code remains. The resulting minimized code is easier for programmers to determine the real cause of failure. (Zeller and Hildebrandt, 2002)

There are several algorithms that can be employed in the delta debugging. One of the simplest approach is based on the concept of binary search. When applied to minimization of program input, the binary search method involves splitting the program input into two halves, each considered as one test case. Each of the test case will be fed to the program and executed, and if the program fails, the test case will be regarded as currently most minimized failing test case, and in turn, it will be further divided into two smaller parts and the process is repeated again. The divide-and-conquer approach will give a failure test case that can no longer be minimized (that still fails the program), which will be the final result of the delta debugging. Figure 2.3 depicts delta debugging flowchart using binary search method.

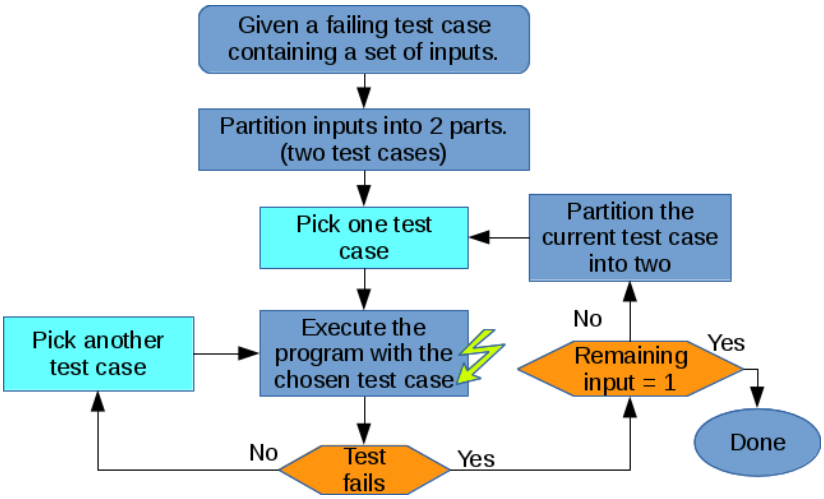

Figure 2.3 Example of delta debugging algorithm using binary search method.

Instead of minimizing a failing test case, it can be used in reverse way, by isolating differences that causes the failure. It does so by maximizing the passing test case. The algorithm is similar to the minimization algorithm, but instead of minimizing the failing test case, it maximizes the passing test case, by collecting all the portions of the program input that are tested to be passed

(originally only portion that produces the failure input is retained). A passing test case with minimum difference from the originally failing test case, is produced. The passing test case can be easily compared to original failing test case to reveal the difference that makes the program to fail.

There are many considerations when designing the algorithm for the delta debugging. The delta debugging may produce a minimized test case with different type of failure from the original, depending on how the algorithm is designed. If the algorithm is designed to be strictly generating minimized test case with exactly same failure, in general the time required would be longer, and the test case will be larger than the absolute minimum.

In addition, binary partitioning might not work for every case since certain failure only occur with a combination of several parts of a test case that may be distributed throughout the test case. For this reason, other algorithms of partitioning needs to be employed, with varying granularity, may need to be employed. A good delta debugging algorithm may also need to dynamically change the partitioning method or employ hybrid techniques to suit a variety of test case scenarios.

2.3.5 Replay-Based Debugging

More often than not, software debugging processes often get very complicated due to the presence of non-determinism in the bug reproduction. In other words, it is not easy to ensure a failure can be reproduced consistently, even when the same set of program inputs are provided, expected failures do not occur every time. This is due to the nature of the program execution itself, which is highly dynamic. Many factors, which includes timing of user actions, can affect code execution of the program in seemingly mysterious ways.

A family of debugging techniques that are called the replay-debugging techniques are currently available to deal with non-deterministic failures effectively. Basically it is a technique that produces program execution that is similar or identical to the original execution, by making