Design Verification Using Formal Techniques

A Thesis submitted in partial fulfillment of the requirement for the Award of the Degree of

MASTER OF TECHNOLOGY

in VLSI Design

Submitted By

Shinal Sharma
602362033

Under Supervision of

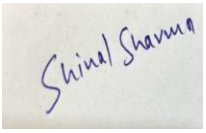
Dr. Amanpreet Kaur
Associate Professor
TIET, Patiala

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
(A DEEMED TO BE UNIVERSITY), PATIALA,
PUNJAB, JUNE, 2025

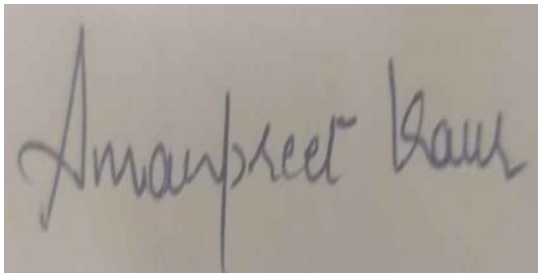
DECLARATION

I, **Shinal Sharma** hereby declare that the work presented in this thesis entitled **"Design Verification Using Formal Techniques"** in partial fulfillment of the requirement for the award of degree of **Master of Technology (VLSI Design**) submitted at **Electronics and Communication Department** , Thapar Institute of Engineering & Technology (Deemed to be University), Patiala is an authentic record of work carried out under supervision of **Dr. Amanpreet Kaur (Associate Prof., Electronics and Communication Department. Thapar Institute of Engineering & Technology, Patiala) and Industrial Mentor Manu Singh (INTEL CORPORATION)** from **June 2024** to **June 2025**. The matter presented in this has not been submitted either in part or full to any other university or institute for the award of any other degree.



Shinal Sharma

602362033



Dr. Amanpreet Kaur (Associate Professor)

Department of Electronics and Communication Engineering

Thapar Institute of Engineering & Technology, Patiala, Punjab

INTEL Tech. INDIA PVT. LTD.

Banglore , India

CERTIFICATE

Date: 24th June, 2025

This is to certify that the Design project report entitled **"Design Verification Using Formal Techniques " is** in partial fulfillment of the requirements for the award of the Degree of **M.TECH** in VLSI DESIGN and submitted to the Department of Electronics & Communication Engineering at TIET, Patiala is a record of bonafide work carried out by **Shinal Sharma**, Roll No. 602362033, during a period from **24 June 2024 till 23 June 2025** under the supervision of **Mr. Manan Desai**, Engineering Manager Verification, Intel Technology India Pvt. LTD. This is to certify that the candidate above statement is correct to the best of my knowledge.

MR.MANAN DESAI
Engineering Manager Verification
Intel Technology India Pvt. LTD.

ACKNOWLEDGEMENT

Feeling gratitude and not expressing it is like wrapping a present and not giving it and I am thankful to all who made it possible. This humble endeavor bears the imprint of many persons who were in one way or the other helped for the completion of this thesis. With deep gratitude, I acknowledge all those guidance and encouragement, which served as a beacon of light and crowned our efforts with success.

I would like to thank **Prof. Kulbir Singh** for giving such an opportunity of internship at Intel Technology India Pvt. Ltd. extend my deepest gratitude to **Mr.Manan Desai** ,Engineering Manager Verification , Intel Technology India Pvt. Lt. for the valuable support he provided me. I am thankful to our PG coordinator **Dr. Bharat Garg** for ensuring that the report submissions and the presentation dates were well fixed such that it would least affect our work at Intel Tech .Pvt. Ltd. I owe my heartfelt gratitude to my guide **Dr. Amanpreet kaur** valuable guidelines, constant assistance, support and constructive suggestions in the betterment of the thesis, without which this would not have been possible. Also, I would like to express my sincere gratitude to my mentor **Mr.Manu Singh** ,Intel Technology who has always supported me in taking new initiatives, accomplishing goals and comprehensive learning. Thanks a lot, to my mentors for their valuable guidance during the report work, they have given me valuable advice and support which has helped me in understanding the technical aspects of the report.

Last but not least, I want to express my gratitude to my family, friends, and colleagues for their prompt assistance and in sightful suggestions.

Shinal Sharma
602362033

ABSTRACT

The increasing complexity of chips has made traditional verification methods more difficult and expensive. To tackle the challenge of accurately implementing intricate designs, current research is exploring the integration of formal techniques with adjustments to design methodologies. It has been suggested that formalizing abstract models early in the design phase can help detect design errors and reduce the cost of fixing bugs. Recognizing that different verification issues require unique strategies is crucial for effectively applying formal verification in the initial stages of design. Each perspective offers a distinct way of reasoning to answer the question, "Why is the design correct?" By employing various models and tools for each perspective, a set of viewpoints can capture the design intuition. This approach allows the models to be sufficiently small for quick construction, validation, and modification. Identifying corner case issues early in the design process results in lower redesign costs compared to discovering bugs later on. Additionally, this thesis includes efforts to cut the number of test cases in half, thereby saving simulation time. The conclusion and future directions of the work are discussed at the end of the thesis.

.

TABLE OF CONTENTS

LISTS OF FIGURES

ABBREVIATIONS

ADC	Analog To Digital Converter
BG	Block Guide
CRC	Cyclic Redundancy Check
DAC	Digital To Analog Converter
DMA	Direct Memory Access
DV	Design Verification
EDA	Electronic design automation
CPU	Central Processing Unit
FPGA	Field Programmable Gate Arrays
FSM	Finite State Machine
HDL	Hardware Description Language
LSB	Least Significant Bit
IC	Integrated circuit
I/O	Input/Output
IP	Intellectual Property
MSB	Most Significant Bit
NoC	Network on Chip
PLL	Phase Locked Loop
RAM	Random Access Memory
ROM	Read Only Memory
RTL	Register Transfer Level
SDV	Software driven verification
SoC	System on Chip
SRAM	Static random-access memory
SVA	System Verilog Assertions
TB	Testbench
UVM	Universal Verification Methodology
VC	Virtual Component
VHDL	VHSIC Hardware Description Language
vPlan	Verification Plan

CHAPTER1
INTRODUCTION

1.1 BACKGROUND

A System on Chip (SoC) is an integrated circuit framework that incorporates reusable intellectual properties (IPs), various controllers, storage components like random-access memories (RAMs) and read-only memories (ROMs), as well as embedded processors. These processors can be categorized as either general-purpose or special-purpose, depending on the requirements and bus architecture topologies for interfacing with different components and ICs. SoCs also include mixed-signal blocks such as digital-to-analog converters (DACs), clock circuitry like phase-locked loops (PLLs), and test designs, all interconnected to perform specific functions. Figure 1.1 illustrates the block diagram of a typical SoC architecture[1].

The primary advantage of utilizing SoCs is the reduction in device size and significant decrease in power consumption. This feature enables the development of various portable devices that maintain their capabilities and functionality without compromising on performance, thanks to these integrated circuits. Consequently, SoCs preserve their attributes and are extensively employed in nearly all electronic devices..

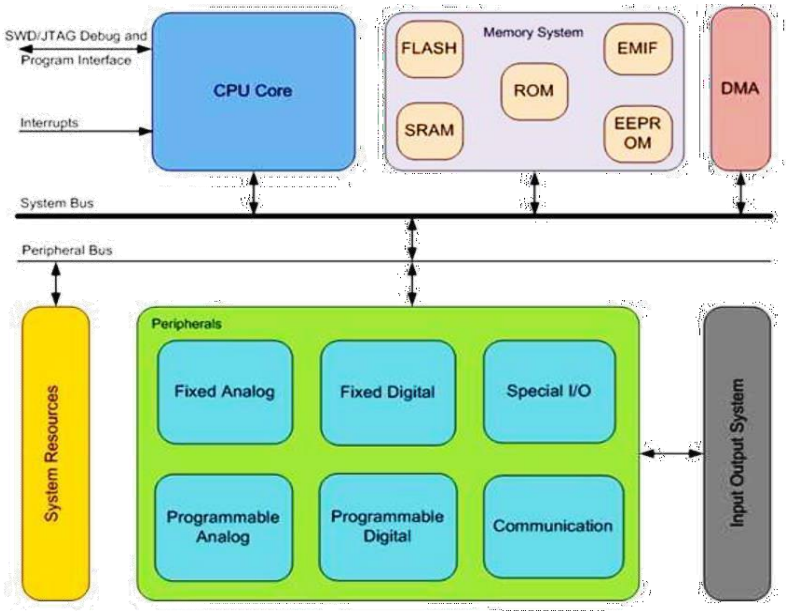


Figure1.1: SoC General Architecture

1.2 SOC DESIGN CYCLE

The initial phase of the SoC product cycle involves gathering client requirements, where the client could be a colleague or a subcontractor. Once the customer specifications are received, they are documented and reviewed. Subsequently, the architectural team determines how to execute the design to meet the specified requirements[3]. As development progresses, the fundamental design criteria outlined in the block guide (BG) may evolve, and design verification (DV) engineers must be prepared to accommodate these changes. After the verification process is completed and the RTL code is free of bugs, the final design is sent for logic synthesis.

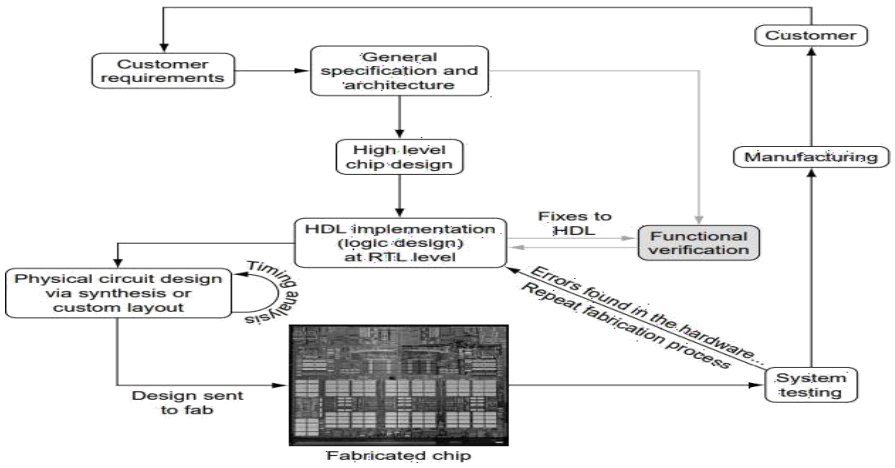


Figure1.2: SoC Design Cycle

The input for the logic synthesizer is the thoroughly verified HDL code, from which a gate-level net list is generated. Essentially, a net list illustrates the connections between logic components and the interpretation of design sub-blocks. Following the critical synthesis stages, the next step in the SoC product cycle is the placement and routing of the design on the silicon die. EDA tools play a vital role at the backend, as they are responsible for estimating timing, power consumption, and the overall chip area of the design[4]. The design is then sent to a fabrication lab for testing before being delivered to the customer. Finally, the product undergoes packaging and testing using real-time programs that simulate realistic scenarios, with feedback provided to the designers. If any bugs are identified, the chip is re-fabricated, which occasionally occurs.

1.3 REUSING IPS IN SOC

The adoption of system-on-chip (SoC) techniques is driven by three key demands in the semiconductor industry: cost reduction, increased design complexity, and shorter design cycles. A fundamental concept in SoC is the reuse of intellectual property (IP) modules. IP cores are electronic components that have been previously developed and result in hardware chips, which can be reused at the SoC level. Soft IP cores are IP modules that allow users to modify their circuit descriptions, expressed in a hardware description language, to meet specific requirements[6]. Firm IP cores offer more control over the design's performance compared to soft IPs. Hard IP cores, on the other hand, are less flexible, more optimized for timing, and present a black-box approach to the user. They are used for specific application needs due to their fixed layout form. For an IP to be reusable, it must meet all necessary.

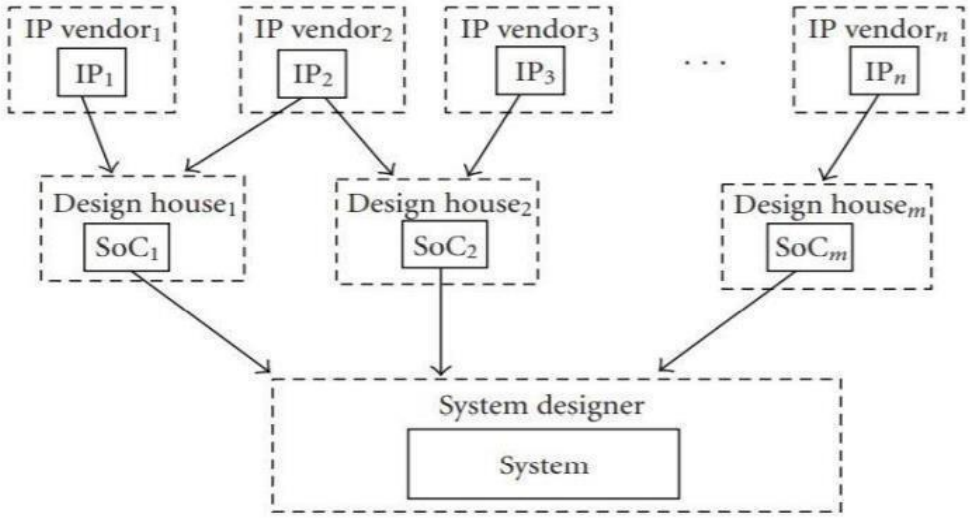


Figure1.3: Reusing IP In SoC environment

In the SoC design process, which involves IP reuse, both software and hardware development are required and occur simultaneously (software/hardware co-design). IPs are chosen based on the SoC application after analyzing the required specifications. During the process, IPs are partially redesigned to enhance their accessibility and reusability at the SoC level, transforming them into virtual components (VCs). This procedure aims to completely overcome interface limitations by creating virtual circuits (VCs) and bus designs. Subsequently, design-related tasks such as voltage level shifters and clock divider placements are addressed, leading to SoC production.

1.4 VERIFICATION AND LEVELS OF VERIFICATION

The verification process for these devices poses significant challenges for companies tasked with meeting customer standards in the production of IP and SoC products. As modern standards grow increasingly complex, they impose a substantial verification burden that can only be addressed through the use of advanced verification techniques and optimized reuse strategies. Notably, verification constitutes over 70% of the total chip design time. It is crucial to resolve all defects and errors in the design before the expensive chip enters production. This principle serves as the core concept and driving force behind verification. The levels of verification are as follows:

1.4.1 IP (Intellectual property)/Module Level

An IP verification engineer is responsible for ensuring that each configuration of the IP is thoroughly checked. This process facilitates the successful integration of RTL at higher levels, such as the subsystem or SoC level. To effectively verify the functionality of the IP, regressive testing generates numerous test cases. IP RTL is typically programmed using parameters in a generalized manner, allowing multiple instances of IPs to exist at higher levels with only these parameters differing. During this stage, the IP is verified within its test environment, known as the IP environment, which is often based on SV/UVM. By the end of the verification process, all design standards should be addressed, and the IP coverage should reach 100%.

1.4.2 SoC Level

RTL codes from various critical and peripheral IPs and platforms are combined to create the SoC (System on Chip) RTL, resulting in a unified code. SoC verification is responsible for ensuring that all I/Os and multiple parameterized verified IPs function correctly together. The SoC verification tests typically include reset test cases, register access test cases, and other tests based on IP functionality.

Key features of interest in SoC verification include:

Connectivity: Verify that the module is correctly connected and that all signals are routed to the appropriate blocks. Assess the block's coverage to identify which ports are protected and which are not. Break down features into individual tests to facilitate their application to other products using the same block.

Parameters: Each module's parameters must be verified. While some parameters can be implicitly tested through connectivity testing (such as bus width), they should always be explicitly mentioned and checked in the edit per item[8].

Functionality at the module/IP level: Integration testing should not focus on functionality at the module/IP level, covering only a few basic functions of the module. This is often inadvertently addressed during network testing.

Synchronous boundaries: Certain modules have synchronous clock domain crossings, interfaces to internal analog blocks, and other features that require testing through timing simulations. Since this cannot be done at the module level, SoC tests and features must be developed[7].

Analog interoperability: If a module is to be directly connected to an analog block at the SoC level, all functional aspects of the interface must be covered by functional tests.

For example, ACRCIP verification includes:

1. Register test cases.
2. Polynomial verification test cases.

Each level has its own verification targets, such as checking all design specifications at the IP level. The primary goal of SoC verification is to ensure the proper integration of multiple designs.

.

1.5 VERIFICATION MANAGEMENT

The verification flow loop is considered complete when the coverage results reach the acceptable percentage outlined in the verification plan[17]. This approach indicates the status of the verification process's completion. The diagram below illustrates the verification management flow.

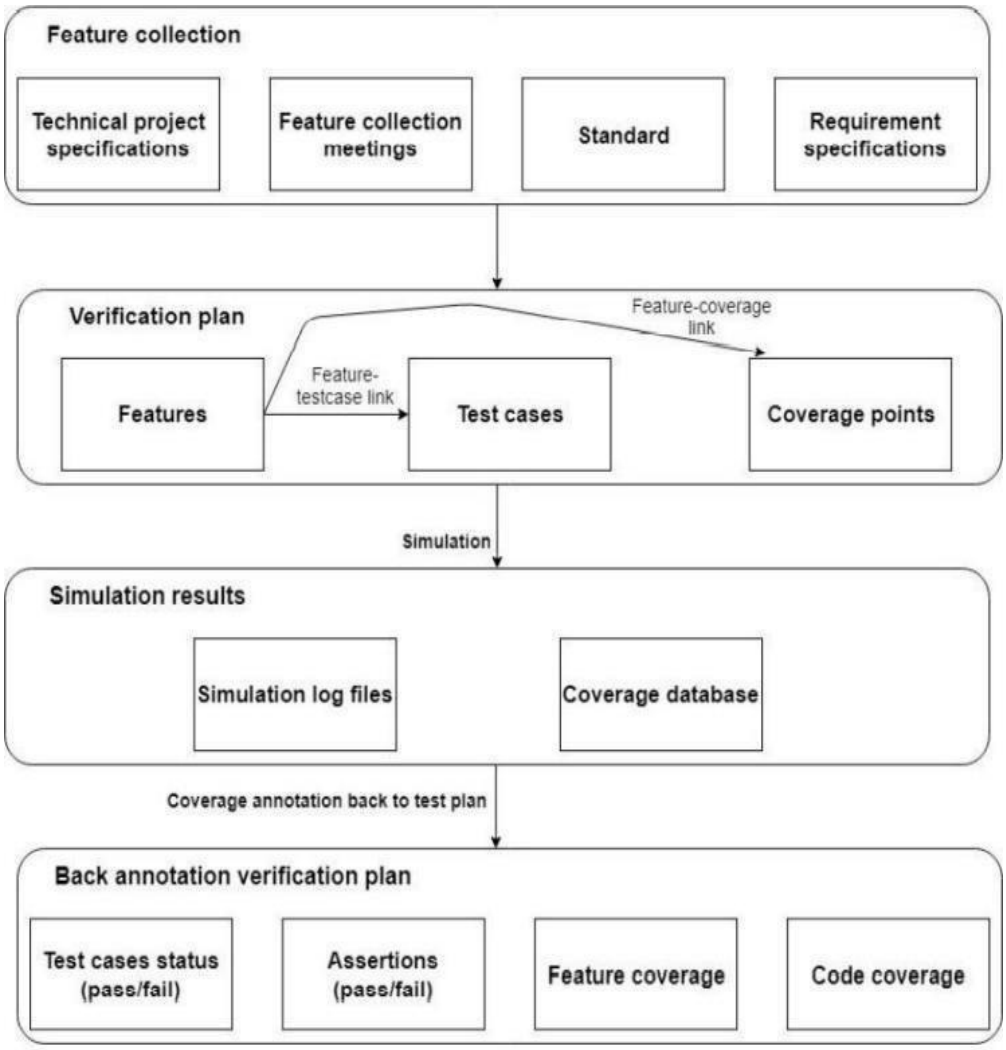


Figure1.4: An Example to Illustrate the Verification Management Flow

The features that surround verification management are:

1.5.1 Feature collection

Verification management is the backbone of the verification process since it organizes the entire verification cycle flow, starting with the formulation of plans and ending with the intended verification outcomes. The verification coverage statistic is then linked to the design features, which consist of design characteristics based on customer needs.

1.5.2 Verification plan

The verification plan, often referred to as vplan [5], is the key document in the verification management process. Typically, a verification plan details the verification strategy by listing all test cases that address every aspect of the design, along with the timeline needed to complete the verification process. Additionally, vplan includes descriptions for each test case. To enhance clarity and organization, it is essential to divide the plan into sections based on each feature, along with its corresponding verification solution, such as test cases.

1.5.3 Verification plan back annotation

When simulations or regressions are conducted, verification tools typically produce coverage reports, simulation logs, and waveforms. Verification management tools, like Cadence's vManager, utilize the results from logs and coverage reports as inputs to generate insights. This approach facilitates back annotation, and the resulting data can be conveniently transferred into an Excel spreadsheet..

1.5.4 Coverage

The Design maturity is evaluated once a substantial level of coverage is attained, making it a crucial metric for functional verification and an integral part of the process. There are three types of coverage metrics: code coverage, functional coverage, and test case coverage. These will be explored in greater detail in the following chapters [5].

The verification team finalizes coverage closure upon reaching the following milestones:

1. All design features must be implemented in the RTL.

2. Every design feature must be verified according to the verification plan.

3. Functional and code coverage must reach 100%.

4. A successive number of regression tests should be clean.

In summary, coverage should be monitored throughout the verification process to ensure visibility into its progress. To maintain high verification quality throughout the cycle, all coverage metrics must be taken into account.

1.6 PROBLEM STATEMENT

As integrated circuit designs and systems become more complex, correctness and reliability checking of these designs becomes more difficult. Simulations-based verification methods are not able to accommodate the increase in scale and complexity of contemporary systems and end up performing incomplete verification and missing design defects. This makes the verification too expensive in terms of design flaws, product release delays, and reliability problems in mission applications.

Formal verification methods, which apply mathematical models and logic to verify the correctness of a design, have become a hopeful solution to this issue. Formal verification methods can give strict guarantees regarding the correctness of a design, such as the lack of errors, correctness of behavior, and safety properties. Nevertheless, formal verification methods are also confronted with some challenges like scalability, expressiveness, and use in practical design flows.

This thesis will seek to investigate and utilize formal verification methods for design verification purposes. More specifically, it will examine the efficacy, limitations, and real-world use of formal approaches to verify intricate designs in hardware systems, software systems, and system-on-chip (SoC) designs. By case studies and experiments, this study will assess the state of affairs of current formal verification tools, their shortcomings, and possible solutions in improving their scalability and compatibility with current verification practices.

The aim of this thesis is to analyze how formal verification methods can successfully be integrated into contemporary design verification processes, and specifically the opportunities and challenges posed by these methods. The investigation will evaluate formal techniques' strengths

and limitations against conventional simulation-based approaches and examine their potential for dealing with scalability, expressiveness, and integration into current design flows. Through the study of case studies of hardware, software, and system-on-chip designs, this research will be able to shed light on how formal verification methods can be used to enhance the correctness and reliability of complex systems. In addition, the thesis will also discuss how the scalability and effectiveness of formal approaches can be improved so that they can be applied in large-scale designs. Lastly, the study will suggest methods for incorporating formal verification with current verification approaches and provide an enhanced and trustworthy verification process for contemporary designs.

This research seeks to contribute to the continued development of formal verification methodologies, with a greater insight into their use in practical applications and the bridging of the gaps to their more general use. By overcoming the existing limitations of formal verification and demonstrating its benefits as a replacement for conventional methods, the thesis will aid the uptake of these methods, producing safer designs and mitigating the risk of expensive errors within crucial systems.

1.7 THESIS ORGANIZATION

The layout of this thesis compiled is as follows:

The organization of this thesis is in a cumulative fashion to lead the reader through the different facets of System-on-Chip (SoC) verification design, culminating in an extensive view of formal and assertion-based verification approaches. Every chapter is dependent on the last one, creating a logical sequence that emphasizes both theoretical foundations and real-world implications of verification in contemporary digital systems.

Chapter 1 is an introduction to SoC world, providing a basic understanding of what System-on-Chip designs entail and the intricacies of designing them. It talks about the general design process, briefly covering the architectural, functional, and implementation levels of SoC design. The chapter highlights the pivotal role that verification must play at every step of the design cycle and introduces different levels of verification, including block-level, subsystem-level, and full-chip verification. It also describes the need for verification planning and management, emphasizing how effective and early verification planning can greatly influence the quality and time-to-market of a chip as a whole. Lastly, the chapter is wrapped up with a brief articulation of the research issue and an overview of the thesis aims, paving the way for the elaborate discussions in the subsequent chapters.

Chapter 2 is dedicated to the literature review conducted to aid the research. The chapter delves into a broad array of work already done in the area of design verification, especially with reference to SoCs. It explores the different verification and validation methods that are widely used in the semiconductor industry, including simulation-based verification, emulation, formal methods, and hybrids. Additionally, it discusses the common verification lifecycle used in industrial settings and describes the challenges encountered at each step of this cycle. These are scalability, completeness, resource shortage, and the complexity issue due to the nature of today's designs. The chapter gives a good background that justifies the interest in using formal techniques for large-scale SoC verification.

Chapter 3 is a detailed overview of various verification techniques available today. It includes traditional and new techniques employed to verify functional correctness, timing, power, and security features of a design. It categorizes the verification techniques into simulation-based, formal, and hardware-accelerated approaches, and describes their respective roles within a verification plan. The chapter also describes how they are combined in realistic verification flows to achieve maximal coverage and efficiency. It describes the tools, languages, and standards used commonly, e.g., SystemVerilog, UVM (Universal Verification Methodology), and formal verification tools. By this, the readers get an understanding of the wide variety of verification methods and the selection thereof according to design demands.

Chapter 4 then moves towards formal verification, a mathematically exact technique applied to establish or refute the correctness of a system relative to a stated specification. The chapter describes how formal methods contrast with simulation, offering deterministic proofs of correctness instead of evidence gathered from test cases. It describes the concepts of formal verification, model checking, and equivalence checking, and how they apply within the context of SoC design. The chapter assesses the strengths of formal methods, like guaranteed coverage and early bug detection, as well as their weaknesses, like scalability issues and specification writing complexity. Particular importance is given to the increasing role of formal verification in large designs, where conventional approaches might lack sufficiency in guaranteeing full correctness.

Chapter 5 presents Assertion-Based Verification (ABV), a formal verification technique used to verify temporal properties of the design using assertions in SystemVerilog Assertion (SVA) language. The chapter describes how assertions are monitors inserted inside the design or testbench that ensure desired behavior dynamically during simulation or statically with the help of formal tools. It defines the types of assertions—immediate and concurrent—and provides their application in real-world examples. The advantages of ABV, including early bug detection, enhanced debugging facilities, and more efficient documentation of design intent, are discussed in detail. The chapter also delves into how ABV is smoothly integrated with simulation and

formal flows, thus proving to be a flexible and robust technique in any verification approach.

Chapter 6 wraps up the thesis with an overview of the work presented and some proposed directions for future research. It looks back at the main findings and results achieved by delving into formal verification methods and assertion-based verification. The chapter suggests avenues for further research, including the automation of assertion generation, formal method integration with AI-based verification tools, and formal verification application in new areas such as security verification and AI accelerators. It also calls for scalable formal tools and methodologies that are capable of matching the growing complexity of today's SoCs. This last chapter highlights the significance of further research and development in verification in order to provide reliability, performance, and safety guarantees for future semiconductor products.

CHAPTER 2

Literature Review

This chapter presents the literature review in context to the types of verification and methodologies available .

2.1 Literature Survey

Y. Kin, J. Yun, N. Kim, and B. Min [5] explored the challenges associated with automation in verification test benches, primarily due to limited reusability. They demonstrated how Python can enhance standardization, reusability, and the quality of verification.

H. Zhaohui, A. Pierres, H. Shiqing, C. Fang, P. Royannez, E. P. See, and Y. L. Hoon [6] presented a practical and efficient SoC verification flow by reusing IP test benches and test cases. They addressed the issue of SoC and IP engineers working in specialized areas and collaborating on debugging based on the SoC-IP interface, which increases time. Their proposed SoC flow successfully reduced SoC verification complexity and debugging difficulty, shortening the verification cycle by half and reducing engineer resources by half.

S. A. Saji and K. Sivasankaran [7] discussed how their test suite addresses challenges in verifying interconnects used in complex SoCs. They also provided an example of writing a test suite sequence and test using the Universal Verification Methodology (UVM), a standard verification methodology based on the System Verilog language.

Manzone, A. Pincetti, and D. De Costantini [8] examined strategies and structures used in the automotive industry to ensure a high degree of fault tolerance for both complete systems and integrated circuits.

Ismail, Azianti, Qiang Liu, and Jung Won [10] discussed system faults and random hardware faults as challenges arising from the increasing complexity and interaction of E/E systems in rapidly growing automobile features within the safety-critical market.

M. K. Wooseung Yang [11] described issues related to verification tools and methodologies for SoC, focusing on planned reuse of IP and pre-verified platforms alongside the co-development of hardware and software. They also reviewed SoC platforms that can be reused and gradually enhanced, as well as platform-based SoC design activities, which can be divided into SoC IP design and integrated system verification.

2.2 Gaps in Study:

The studies referenced primarily focus on improving SoC verification through automation, test bench reuse, and addressing fault tolerance, but they do not fully explore the potential of **formal verification techniques**. For example, **Y. Kin et al. [5]** discuss the automation of verification using Python, which helps with reusability and standardization but doesn't incorporate formal methods that could provide mathematical guarantees of correctness. Similarly, **H. Zhaohui et al. [6]** present an efficient SoC verification flow that reuses IP test benches to reduce verification complexity, but their approach does not consider how formal verification could help ensure completeness and rigor in verifying the SoC's behavior.

In the case of **S. A. Saji and K. Sivasankaran [7]**, they use UVM (Universal Verification Methodology) to verify interconnects in complex SoCs. While UVM is an industry-standard methodology based on SystemVerilog, it is still a simulation-based approach, which may not be exhaustive. **Manzone et al. [8]** discuss fault tolerance strategies for automotive systems, but their focus is mainly on traditional error detection methods rather than the exhaustive analysis that formal verification can provide. Similarly, **Ismail et al. [10]** explore system and hardware faults in automotive E/E systems but do not consider how formal methods could help in systematically identifying corner cases and ensuring correctness under all conditions.

Lastly, **M. K. Wooseung Yang [11]** discusses the reuse of IPs and platforms in SoC design but does not touch on how formal verification techniques could ensure the correctness of these reused components, especially as designs grow more complex.

The gap in these studies is clear: while they make important contributions to enhancing the efficiency, collaboration, and fault tolerance in SoC verification, they do not sufficiently address the role of formal verification techniques. Formal methods such as model checking, theorem proving, or formal specification could address issues like exhaustive coverage, scalability in large designs, and ensuring correctness across all possible scenarios. These methods could fill the current gap in traditional verification methodologies by providing provable correctness and ensuring that all potential behaviors of the system are validated.

2.3 Objective of the thesis :

The objective of this thesis is to investigate the application of formal verification techniques in the design and validation of System-on-Chip (SoC) architectures. This research aims to explore how formal methods, including model checking, equivalence checking, and theorem proving, can be effectively utilized to ensure the correctness, performance, and security of SoC designs. The thesis will focus on identifying the specific challenges associated with SoC verification, such as the complexity of multi-core architectures, diverse communication protocols, and hardware-software interactions. Additionally, the work seeks to evaluate the efficiency and scalability of formal verification tools in handling large, intricate SoC designs and propose solutions for their integration into current design workflows. Ultimately, this thesis will contribute to the advancement of formal verification practices for SoCs, enhancing their reliability and reducing the risk of errors in modern embedded systems.

CHAPTER 3

Types of Verification and its Methodologies

The purpose of verification is to determine whether the design (RTL) is functional. A design verification engineer requires a test bench framework to assess if the design behaves as expected. Consequently, the verification process is crucial before a chip is sent for manufacturing and ultimately delivered to the end customer. This process accounts for 70% of the total design cycle time. If functional errors are discovered after the chip has been taped out, it results in financial losses, as the design cycle must be restarted[2].

Below is a basic test bench architecture, essential for verifying the design at a minimum level. It begins with stimulus generation (inputs from the test bench), where the design under test (DUT) can be directed or randomized[20]. The primary benefit of using randomized stimulus as input is the enhancement of coverage metrics, as the design's terminal conditions or corner case scenarios are better defined. However, when using random stimuli, it is important to ensure they are not unconditional, meaning the selected input must have a value that qualifies it as a valid design input.

.

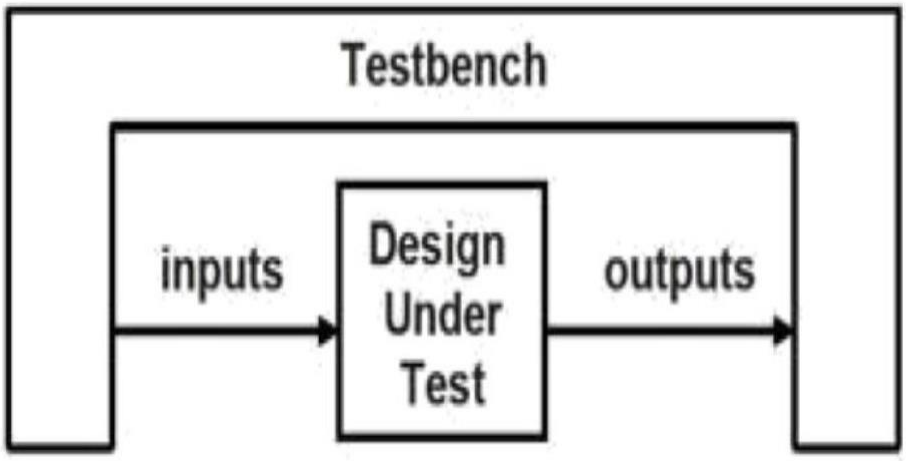


Figure 3.1: Basic Test Bench Structure

3.1 TYPES OF VERIFICATION

3.1.1 Direct testing verification

The RTL designer documents all the specifications of the IP and provides them to the verification engineer as direct stimulus. The verification engineer reviews this document, known as the Block Guide, and develops a verification plan based on it. Following the vplan (verification plan), test cases are systematically created to cover the design's functionality, progressively achieving 100 percent functionality coverage[16].

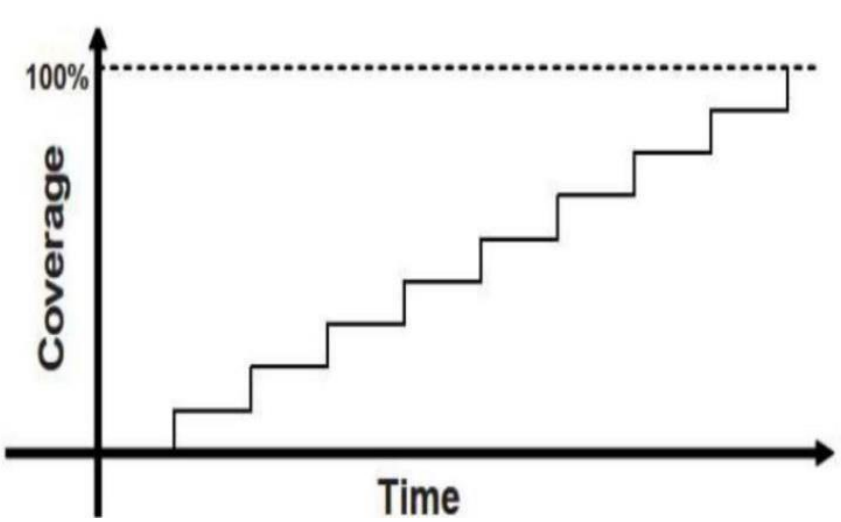


Figure3.2: Coverage Vs Time in Direct Stimuli Verification

3.1.2 Constrained Random Stimulus Verification

A test is considered randomized when it assigns a completely different value to a variable. Consequently, the variable is declared as rand in the test. The simulator generates a random seed each time the variable is randomized, assigning various values to that variable, which enhances the coverage of the DUT[19]. Repeating the same test multiple times yields different results.

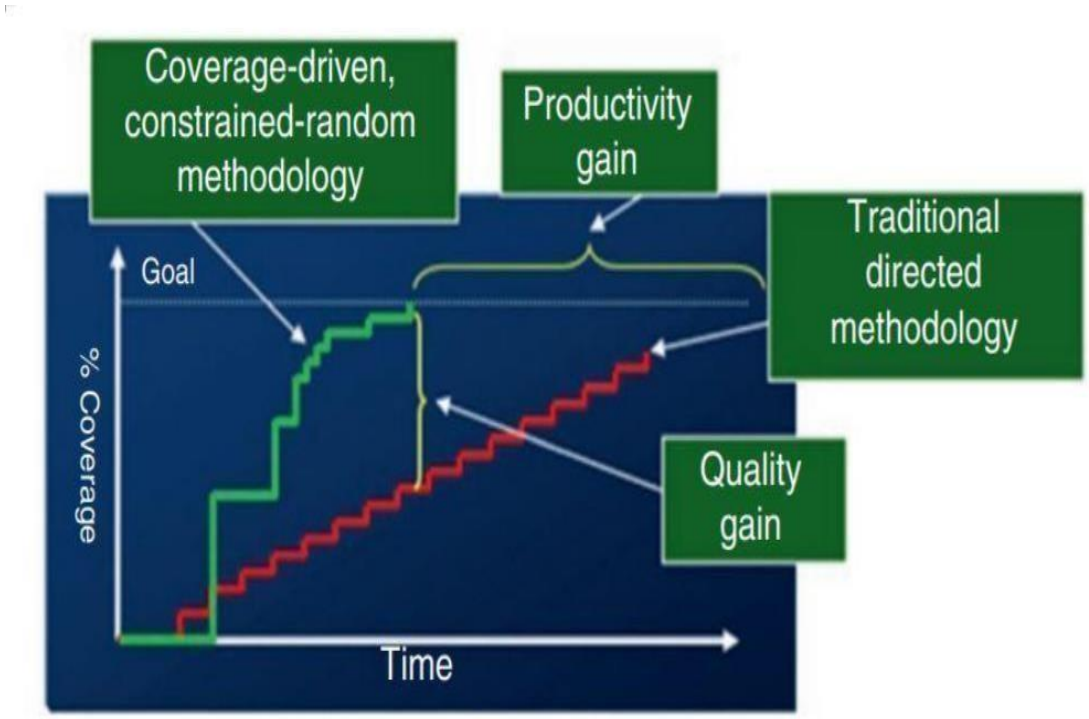


Figure3.3: Coverage Vs Time in Constraint Random Verification

3.1.3 Formal verification

The application generates random stimuli and compares the outcomes to System Verilog Assertions (SVA). SVA is a subset of the System Verilog language that enables users to specify higher-level design assumptions. An assertion instructs the verification tool to examine a property, which represents the design intent specified by the user. Consequently, properties are expressed as assertions, and these verification requirements must be satisfied during simulation.

Assertions simplify debugging, especially when addressing complex design issues. They can be managed and toggled on or off as needed. For instance, if assertions are necessary for a specific test, they can be activated, while remaining inactive for other tests[3].

Since no single technique can fully evaluate a device under test (DUT), the industry employs a combination of these techniques. This approach allows the design to be validated by writing SVA when constraint random stimulus verification is used, accommodating the introduction of minor functionalities to the design in the future.

.

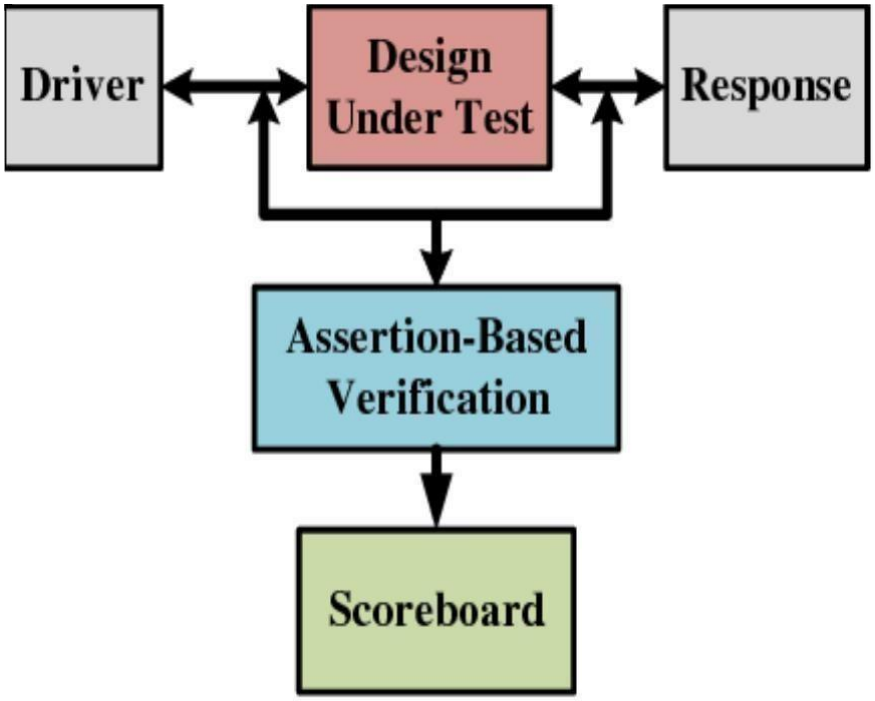


Figure3.4: Formal based verification

3.2 ASSERTION BASED METHODOLOGY

System Verilog Assertions are essentially checks to determine whether a design meets specified criteria. An assertion fails if certain attributes of the simulation do not behave as expected, or if prohibited conditions arise during the simulation. This allows the designer to express their interpretation of the standard[14].

3.1.4 Immediate Assertions

These assertions are used to verify the current conditions. They are non-temporal, meaning they are executed within procedural blocks (such as initial/always and tasks/functions) rather than over time or clock cycles. An assertion fails if the expression evaluates to 0, X, or Z.

Syntax: Assert (expression) [pass_statement] else [fail_statement] Example: always @ (posedge clk) begin

if (state==REQ) begin //if current state is REQ

assert (req1 || req2) //Check whether req1 or //req2is high

$info (“Correct State”);

else $error (“Incorrect State”)end

3.1.5 Concurrent Assertions

These assertions evaluate a sequence of events that occur over multiple clock cycles, making them temporal. Concurrent assertions are defined using the "property" keyword, which represents a design standard that must be verified. They are termed concurrent because they occur simultaneously with other design elements.

Syntax: Assert property (expression) [pass_statement] else [fail_statement] Example:

property req_ack; @ (posedge clk) req ##2 ack ##1 !req ##1 !ack; end property assert property(req_ack) else $error(“req_ackpropertyviolated”);

The number of clock cycles is indicated by a number. Violation will be detected if ack is not high two clock cycles after req is high. If req and ack arrive on time, but reqis not low in the next clock cycle, violations will occur.

3.3 METHODOLOGY USED TO VERIFY SOC

The software-driven verification (SDV) approach is employed, where the stimulus is written in C or assembly code and executed on the core to test the functionality of the DUT. Concurrently, System Verilog sequences defined as stimuli can be utilized to manage test bench components, such as drivers and monitors. The development of test cases involves writing them in C and using the Verilog test bench[8].

A "Porthole mechanism" is provided in the simulation environment to facilitate synchronization between the C test code executed by the ARM processor and the Verilog test bench. This mechanism allows the memory map to use specially reserved addresses. When these addresses are written to from the C test, the Verilog environment detects a change in the address bus, displays a predefined message, and triggers a predefined Verilog event [11] based on the address

and data value sent. With the development of more directed test cases, there is a strong emphasis on C-driven verification. Vertical reuse from module/IP to SoC level verification components/stimuli is not anticipated due to the differing methodologies and processes involved in designing such components/stimuli.

.

3.1.6 ASSUMPTIONS BEFORE SDV

Before implementing SDV, several assumptions must be taken into account, including the following:

- A key assumption during SoC verification is that all modules/IPs are provided after undergoing comprehensive validation at the module/IP level.
- The feature list is verified using the verification and block guide for each module/IP.
- Based on this information, the set of IP features that need to be covered during SoC verification is provided to the verification owner.
- In addition to these features, the primary focus of this verification will be on achieving 100% toggle coverage for all integrated modules.

3.1.7 ADVANTAGES OF SDV

Software-driven verification offers several advantages, some of which are outlined below:

In software-driven verification, processors are retained within the DUT and used as part of the verification strategy. Software is developed to run on these CPUs to test the hardware, and this application can be created either manually or automatically. By 2014, use cases had become widely recognized as a method for describing verification scenarios and developing specialized test cases, with limited random approaches occasionally employed. Commercial applications utilize graph-based methods to capture these use cases, although no standard has been established yet[13].

Another notable aspect of software-driven verification is that processor models can exist at various levels of abstraction, all of which have traditionally been register accurate and object code compatible. This means that software written for an actual processor can run on an instruction-set accurate model, an abstract model, or a model mapped onto an emulator. This facilitates test portability throughout the entire development cycle, from virtual prototyping to simulation, emulation, FPGA prototyping, and finally to

actual silicon [12].

In SoC processes, we typically focus on transaction initiation control. The configuration of dynamic memory access (DMA) channels for data transfer, the initialization/configuration of registers for various blocks/modules, and the handling of interrupts and other exceptions are all managed via the processor. These configurations and transactions are executed at the SoC level when user-written C or Assembly Language code is converted to hex code (machine code) and executed by the processor. The boot loader [13] is responsible for loading the image/.hex into memory (SRAM).

3.1.8 C CODE TO HEX FILE CONVERSION

The execution of C code involves four distinct steps, each utilizing a different tool: the preprocessor, compiler, assembler, and linker. The process begins with the preprocessing phase, which involves executing macros, including files, and handling conditional compilation directives. If no "make errors" occur during this phase, the process advances to the compilation stage. Provided there are no compilation errors, assembly files are generated at this point. During the assembly phase, the assembler converts these assembly files (.asm) into object files (.obj). In the linking stage, the linker transforms the object files into an executable format (.exe/.elf). Finally, the executable file (.exe/.elf) is converted into a .hex file, which is loaded into SRAM memory and executed by the core. The entire process is illustrated in the diagram below.

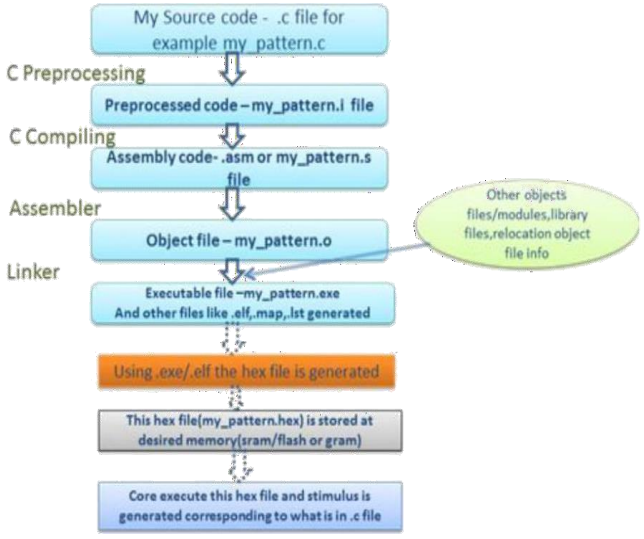


Figure 3.5 C to Hex File Conversion

3.4 VERIFICATION CYCLE

The cycle is divided into four phases, as follows:

1. Development: This includes the DV architecture, verification method, test bench, and tests.

2. Simulation: This comprises, among some other things, successful RTL compilation, elaboration, and waveform synthesis.

3. Debugging: This involves debugging at the signal, transaction, and other levels, and it becomes a substantial challenge if assertions are not employed in the test bench.

4. Coverage: This comprises functional, code, and SVA coverage analysis, which feeds back to the development stage in phase 1.

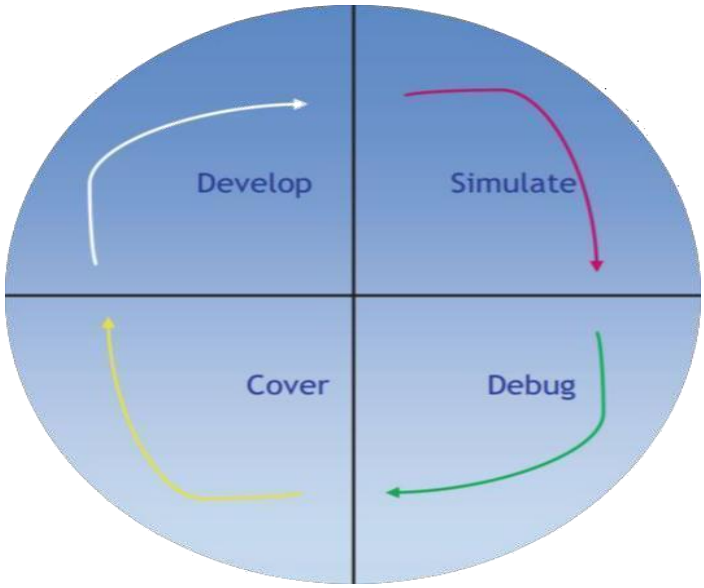


Figure3.6: Typical Design Verification Loop

Minimize time to debug, develop, improve robustness, reduce the amount of time to simulate and increase efficiency, effective design coverage, and other are some of the primary issues that each of the four stages described above must face as they progress through the design verification cycle..

3.5 VERIFICATION CHALLENGES

As modern devices become increasingly complex and time-to-market pressures intensify, engineers face greater demands to complete verification tasks within significantly shorter timeframes. Several challenging situations must be navigated to address the growing complexity of designs and successfully complete a meaningful verification project. Here are some examples of such challenges:

Verification Productivity: This refers to the ability to execute larger design verification plans in less time. Design engineers have achieved efficiency gains by progressing from transistor-level to gate-level, RTL, and ultimately system-level design approaches. Consequently, verification engineers must apply similar efficiency improvements to manage larger devices. Achieving substantial efficiency gains involves advancing to higher levels of abstraction, whether through verification tools or the use of pre-validated functional blocks[8].

Verification Efficiency: This pertains to the amount of human intervention required to complete a verification task. As design complexity increases, manual involvement is expected to be minimized. Enhancing verification efficiency is argued to require increased automation in the verification environment through effective tool techniques, ultimately reducing manual intervention.

Verification Reusability: This is the ability to reuse an existing verification environment. Addressing reusability involves adapting the current verification framework for a new project or similar tasks. Thus, constructing a modular architecture for the verification environment is key to reusability[18]. In this context, module boundaries are identified as components that can be reused in other projects, and detailed descriptions of the verification architecture can enhance code quality.

Verification Completeness: This framework aims to encompass as much design functionality as possible. By allocating more time to improve verification completeness, productivity, efficiency, and reusability are enhanced. The focus is on efficiently handling the functionality of large designs using current verification tools and methodologies.

In summary, these verification challenges must be addressed through modular design techniques and diverse methodologies that automate manual tasks. For instance, did our testing cover all relevant issues, and if not, what needs to be adjusted? Updated documentation is one example of necessary changes.

CHAPTER 4

Formal Verification of SoC Design

4.1 INTRODUCTION

Formal verification is a process that involves verifying the correctness of a design implementation against its specifications using mathematical theories. A temporal formula, which captures specific design behavior, is thoroughly checked against the mathematical model of the design for all valid input values. Commercial formal verification tools support the use of languages with enhanced syntax in industrial settings, simplifying the definition of temporal formulas. These temporal formulas, also known as properties, are manually derived from specifications. If the mathematical model does not support the temporal formula, an error trace, often called a counterexample, is provided[20]. When the model satisfies the formula, it demonstrates that the design behaves according to the specifications. The three primary categories of formal verification methods are Model Checking, Equivalency Checking, and Theorem Proving. The formal techniques are categorized as illustrated in Fig. 4.1.

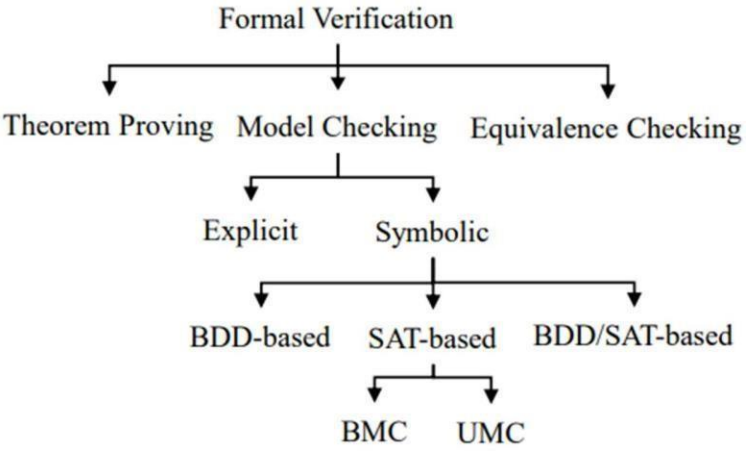


FIGURE 4.1 Formal Verification Techniques.

4.1.1 Theorem Proving

The subfield of formal verification known as "theorem proving" focuses on automating formal reasoning using the principles of logic. In theorem proving, the system is represented through a series of mathematical definitions by applying the rules of mathematical logic. Theorems derived from these definitions are intended to represent the expected attributes of the system. Theorem provers utilize first-order and higher-order logic provers to verify system behavior. First-order logic is considered the most semi-decidable and comprehensible form of logic, and any logic more expressive than first-order logic is undecidable. To effectively employ first-order logic for hardware verification, natural numbers are used to simulate time, particularly for sequential circuits. However, first-order logic lacks comprehensive formalisms for natural numbers. It has been demonstrated that higher-order logics can be applied to hardware verification. Due to the undecidable validity of higher-order logics, proof systems are interactive and typically serve as proof assistants. As a result, theorem provers must be used alongside other formal techniques because they do not offer fully automated procedures, even though they can be applied to evaluate reactive digital systems.

4.1.2 Equivalence Checking :

Equivalence Checking, or Formal Equivalence Checking (FEC), involves using mathematical reasoning to determine the equivalence between two model representations. These representations can be derived from two different designs expected to consistently produce the same outputs, from the same design using different platforms (such as Verilog or VHDL), or at varying abstraction levels (such as RTL or gate level). In the industry, two common types of FEC are sequential equivalence checking and combinatorial equivalence checking[19].

Combinatorial equivalence verification is used to compare two iterations of the same design (or circuit) at different abstraction levels, such as verifying whether a synthesized netlist matches its RTL description. This process involves state matching to identify comparable state variables between the two circuits, followed by an equivalence check. During state matching, all state variables are grouped into a single equivalence class, and

non-equivalent state variables are identified, leading to the division of equivalence classes. Techniques based on satisfiability (SAT), automatic test pattern generation (ATPG), binary decision diagrams (BDD), and structural and logic modeling are employed to demonstrate non-equivalence among state variables. BDD-based techniques do not scale well for large designs, and SAT-based methods for equivalence checking are considered challenging due to their extensive re-convergent fan-out structures.

Sequential equivalence checking is used to verify whether two models produce the same set of outputs at all time points for an equivalent set of inputs. It is not necessary for the internal nodes of the designs to be equivalent to achieve sequential equivalence. The goal of these equivalence tests is to determine if a set of characteristics consistently yields the correct values for a design's outputs.

4.1.3 Model Checking :

Model checking, also known as property checking, is an algorithmic approach used to demonstrate that the behavior of a sequential system aligns with its design. It is the primary method employed by formal verification tools to analyze the behavior of a design implementation. By evaluating the validity of temporal formulas against the mathematical model of the implementation, a model checker verifies that the system's implementation adheres to its specifications[17].

To ensure that a design implementation meets the design specification, a model checker requires the following components:

1.A mathematical model of the implementation with sufficient expressiveness,

2. A suitable specification language to define the expected design behavior, and

3. An effective proof method (algorithm).

Explicit model checking involves explicitly describing every possible state of a design. However, due to state space explosion, explicit model checking is nearly impossible for architectures with moderate to high complexity.

4.2 Functional Verification Process

Once design specifications are established for a processor design project, hardware designers begin constructing a detailed, functional representation of the design using a hardware description language (HDL), such as Verilog or VHDL. Verification engineers create a verification environment to achieve the objectives outlined in the verification plan. Various design blocks are assigned to teams of design engineers who build according to specifications, while verification engineers perform unit-level functional verification. As design blocks become available, they are integrated into subsystems and undergo verification. This process continues until all design blocks are completed and integrated into the system at the highest level. Design and verification engineers work together to identify and resolve bugs discovered during verification. Pre-silicon verification refers to the functional verification process conducted before constructing a silicon prototype[16]. During pre-silicon verification, the HDL representation of the design is systematically compared to the specifications using the verification environment. Formal verification and, more extensively, simulation-based verification are the primary methods used for pre-silicon verification.

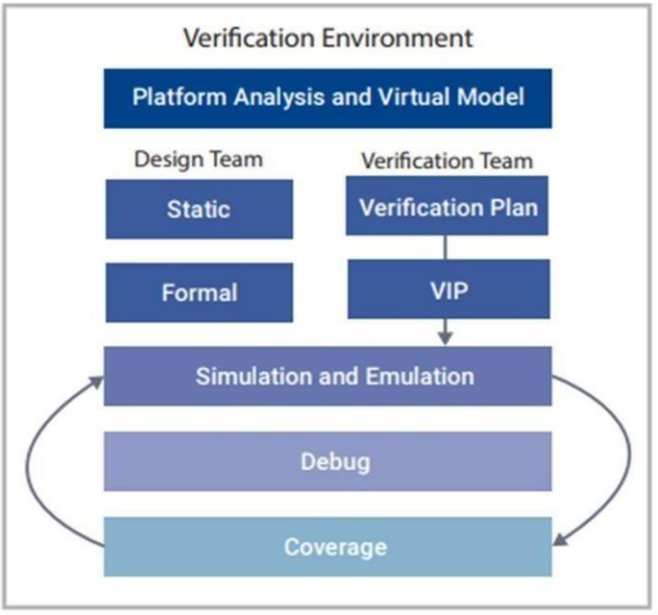


Figure 4.2 High level view of the chip verification flow.

Formal verification tools convert the HDL representation of the design into Boolean functions, which are then compared to attributes derived from the specifications. If engineers can accurately express design specifications as Boolean properties, the tools can mathematically verify or refute the design's accuracy. The strong assurance formal verification provides when the design is correct, along with the counterexamples it generates when the design is flawed, makes it a highly desirable verification tool. However, fully capturing a design's specifications as formally provable attributes is a challenging task. Additionally, the computational demands of formal verification tools are so high that they only permit the analysis of small design units and subsystems, limiting their effectiveness to smaller design blocks. Since the goal of this dissertation is to verify the accuracy of complex designs, we will not explore formal verification further[12].

Simulation-based functional verification constitutes the majority of verification work performed before a chip's physical implementation. Software-based simulation systems convert the hardware description of the design into an executable file compatible with any computer. The verification testbench, a software running alongside the simulated design, directly interacts with the simulation and has unrestricted access to all design signals. The flexibility of software simulation and the extensive visibility into design signals provide robust verification and troubleshooting capabilities. Engineers can develop advanced checkers, analyze the design using debuggers, and capture waveforms, among other tasks. Unfortunately, software simulators are slow when handling complex designs, typically simulating only hundreds of design cycles per second. Simulating one second of a design's operation at these rates could take several months.

.

Figure 4.3: Approaches of Verification

4.2.2 Simulation Based Verification :

Simulation-based verification is a commonly employed method for verifying designs in software development. This approach involves subjecting the design to a test bench, applying input stimuli, and comparing the output against a reference output. The test bench can be either pre-produced or generated during the simulation process, while the reference output can be created either in advance or in real-time.

The effectiveness of modeling a test on a design is determined by the level of coverage the test provides. Coverage tools generate reports on code or functional coverage, enabling designers to identify untested components and develop tests to address those areas.

When a bug is identified, it is essential to inform the designer and resolve the issue. This can be accomplished by logging the bug into a bug tracking system, which sends notifications to the design owner. The system tracks the bug's progress through various stages, including being opened, validated, fixed, and closed.

The typical flow of simulation-based verification is illustrated in Fig 4.4.

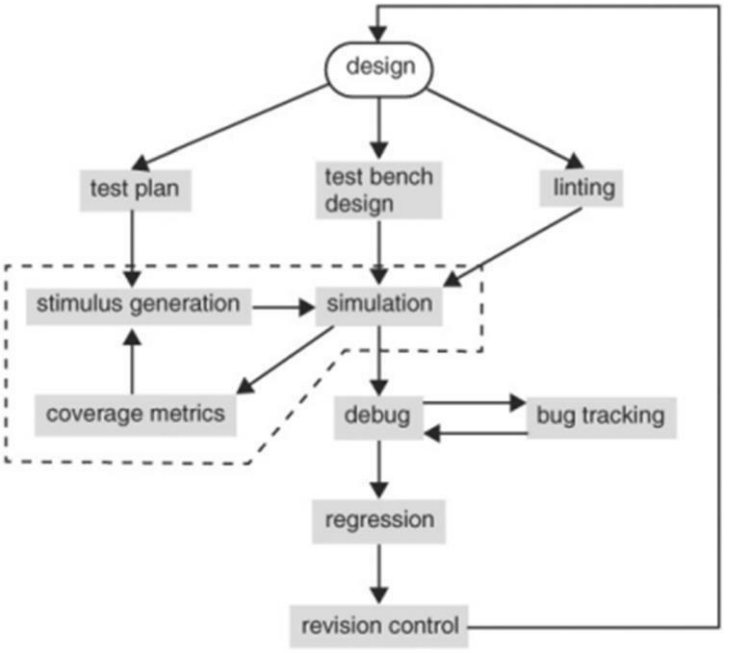


Figure. 4.4 Flow of simulation-based verification

4.2.3 Formal Method Based Verification :

Formal verification provides properties on mathematical models that represent the system implementation. Specifically, model checking tools are used to identify design errors. Temporal logic properties are defined to formally verify the correctness of a design implementation against its specifications.The formal method-based verification approach differs from simulation-based methodology in that it does not require the generation of test vectors, although it is otherwise similar. Formal verification can be categorized into two main types: equivalence checking and property verification[14].

Equivalence checking involves determining whether two implementations are functionally identical. Using a simulation-based approach for equivalence checking is impractical due to the virtually infinite number of vectors in the input space. In formal verification, the result of an equivalency checker is clear-cut. However, industrial equivalency checkers have not yet become fully automated solutions and often require users to identify corresponding nodes in the two circuits to narrow the search area for the checkers. The nodes identified by users as equivalent are known as cut points. Sometimes, a checker can deduce equivalent nodes from their names.The other type of formal verification is property checking[13]. Property verification entails analyzing a design and a property, which acts as a partial specification of the design, to determine if the design meets the property. A property serves as a redundant explanation of the design, effectively confirming its validity. A program that verifies a property is commonly referred to as a model checker, as it involves representing a real circuit in the form of a computer model.The typical flow of formal-based verification is illustrated in Fig 4.5.

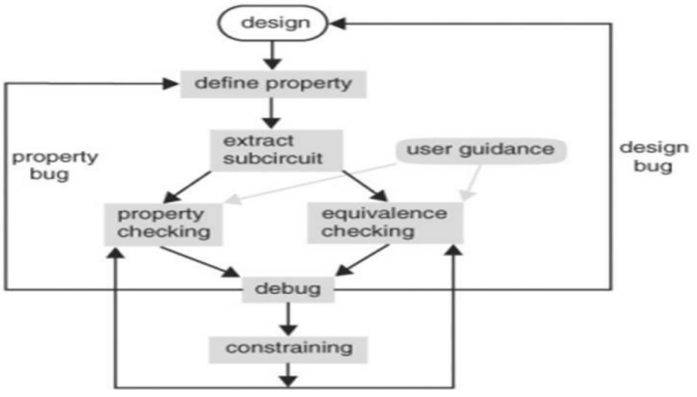


Fig 4.5 Flow of formal based verification

4.3 Simulation-Based Verification versus Formal Verification

The primary difference between formal verification and simulation-based verification is that the latter does not require input vectors while the former does. In simulation-based verification, producing input vectors first and then deriving reference outputs is the mindset. The formal verification procedure involves thinking in the opposite direction. The user first specifies the desired output behavior, which the formal checker is then left to confirm or refute. Users don't give input stimuli any thought at all. The formal technique is output driven, whereas the simulation- based methodology is input driven. The tendency to think input-driven is more common and is mirrored in the perceived difficulty of utilizing a formal checker[14]. Completeness— the absence of any gaps in the input space—is another selling factor for formal verification, while simulation-based verification struggles with this issue. This formal verification's power, meanwhile, might occasionally give rise to the false belief that a design is 100% bug-free after formal verification. To find out if formal verification is accurately understood, let's compare simulation-based verification with formal verification. According to this perspective, input space sampling can be used for verification in simulation-based methods. If every point is not sampled, there is a chance that a mistake will evade verification. From an output standpoint, formal verification verifies a set of output points at a time (a collection of output points constitutes a property); simulation-based verification verifies a single output point at a time[11]. This comparison between formal verification and simulation-based verification is shown in Figure 3.6. Therefore, it must be further demonstrated that the collection of attributes that have been formally validated as a whole forms the specifications in order to fully verify that a design satisfies its specifications using formal methods.

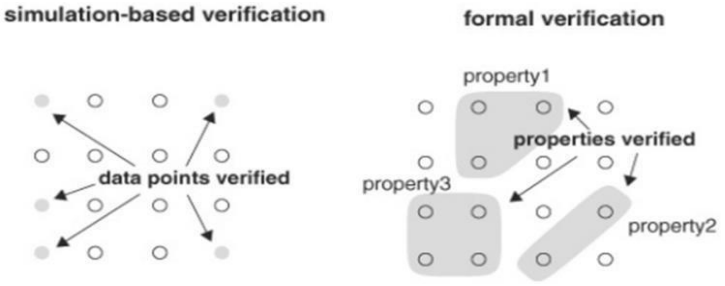


Figure. 4.6 An output perspective of simulation-based verification versus formal verification

Both simulation-based and formal verification techniques are essential for modern system verification. While simulation is widely used and practical, it lacks exhaustive coverage and may miss critical corner-case bugs[8]. In contrast, formal methods provide mathematical certainty but can be computationally expensive and challenging to scale for large designs.

The best approach is a hybrid methodology that leverages the strengths of both techniques—using formal verification for correctness proofs and simulation for practical, real-world testing. As verification tools improve and automation advances, formal techniques will become more accessible, further enhancing design reliability across industries.

CHAPTER 5

Assertion Based Verification

5.1 Introduction

Formal verification conforms that the design meets its specifications. In order to achieve this the first step needed is to find a way to express what it means for a design to be correct. This can be achieved by writing assertions using the System Verilog Assertions (SVA) language. The SVA can be thought of as several layers of increasing complexity as shown in figure 5.1

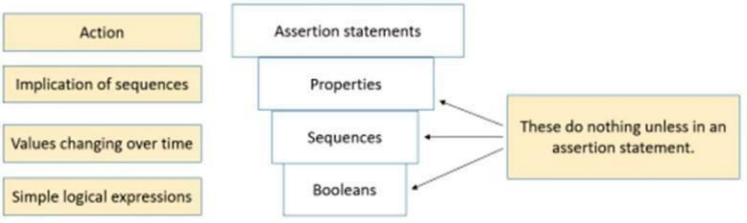


Figure 5.1 Layers of SVA assertion language

5.2 Terminology

Assertion-Based Verification (ABV) is a methodology in digital design verification aimed at enhancing the efficiency and effectiveness of detecting design errors. It utilizes assertions—formal, executable statements embedded within the design or testbench—that define expected behaviors and conditions[19]. ABV aids in identifying functional bugs early in the design cycle, thereby reducing debugging time and improving design quality.

Booleans: These are simple logical standard Boolean expressions, which can be a single logic variable or a Boolean formula such as a & b. Boolean expressions can be used in sequences or properties, as illustrated in Figure 3.7, and are evaluated based on the sampled values of all variables. Sampled values refer to the values of variables at the end of each previous simulation time step.

Sequences: Sequences are statements containing Boolean expressions that occur over time. The simplest sequences are linear, meaning they consist of a list of finite Boolean expressions that occur in linear order over increasing time. The passage of time is defined by a clocking event. Sequences are composed using concatenation, which specifies the time delay with the ## operator, from the end of the first sequence to the beginning of the

second sequence, as shown below:

a ##N b indicates that signal b should be true on the Nth clock tick after signal a was true.

Properties: A property combines sequences with additional operators to capture the expected design behavior based on design specifications. A property can be used as an assumption, assertion, or coverage specification but does not produce results on its own. Below is an example of a named property reqgnt:

property reqgnt;

req |-> s_eventually gnt;

endproperty

The antecedent and consequent in the property are connected via implication operators |-> or |=>. The |-> operator is overlapping, meaning that if there is a match for the antecedent, the endpoint for the match is the start point for evaluating the consequent expression. The |=> operator is non-overlapping, meaning the start point for evaluating the consequent is one tick after the match for the antecedent.

Assertion Statements: An assertion statement is used to validate the behavior of a system. Properties do not produce results on their own; they must be included in an assertion statement using one of the following keywords: assert, assume, or cover.

Assert: Specifies the property as an obligation for the design to verify that the property holds.

assert property (req |-> gnt);

Assume: Specifies the property as input constraints on the environment. Formal tools use these constraints to generate input stimuli.

assume property (!req |-> !gnt);

Cover: Used to monitor property evaluation for coverage, ensuring the intended behavior occurs at least once by finding a single trace for it.

```
cover property (req |-> gnt);
```

Assertions can be of two types—immediate and concurrent:

Immediate Assertion Statements: Simple assertions that follow simulation event semantics and are executed in procedural blocks. They lack clocking or reset and do not support advanced property operators, thus cannot check conditions involving the passage of time. Immediate assertions are defined using only the assert keyword without the property keyword:

```
immediate1: assert (!req && !gnt);
```

Concurrent Assertion Statements: Follow clock semantics and can describe behavior involving the passage of time. They support advanced property statements about logical implementation, including time intervals. Concurrent assertion statements use both assert and property keywords:

```
conc1: assert property (a ##2 req |=> gnt);
```

It is important to note that immediate assertion statements are primarily used for simulation. In this thesis work, only concurrent assertion statements were written to verify the design behavior.

5.3 Design Code

The designs that we have used include the logic of the FIFO and Arbiters.

First- In First- Out (FIFO) Design Code:

```
module fifo #(
    parameter DATA_WIDTH = 8,
    parameter DEPTH = 16
)(
    input logic clk,
    input logic rst_n,
    input logic wr_en,
```

```
    input logic rd_en,
    input logic [DATA_WIDTH-1:0] wr_data,
    output logic [DATA_WIDTH-1:0] rd_data,
    output logic full,
    output logic empty,
    output logic [$clog2(DEPTH):0] count
);
    logic [DATA_WIDTH-1:0] mem [0:DEPTH-1];
    logic [$clog2(DEPTH)-1:0] rd_ptr, wr_ptr;
    assign full  = (count == DEPTH);
    assign empty = (count == 0);
    // Write logic
    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            wr_ptr <= 0;
        end else if (wr_en && !full) begin
            mem[wr_ptr] <= wr_data;
            wr_ptr <= wr_ptr + 1;
        end
    end
    // Read logic
    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            rd_ptr <= 0;
            rd_data <= 0;
        end else if (rd_en && !empty) begin
            rd_data <= mem[rd_ptr];
            rd_ptr <= rd_ptr + 1;
        end
    end
```

```
    // Counter logic
    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            count <= 0;
        end else begin
            case ({wr_en && !full, rd_en && !empty})
                2'b10: count <= count + 1;
                2'b01: count <= count - 1;
                default: count <= count;
            endcase
        end
    end
endmodule
```

First- In First- Out (FIFO) Assertions using SVA Macro's :

```
module fifo_assertions #(
    parameter DEPTH = 16
)(
    input logic clk,
    input logic rst_n,
    input logic wr_en,
    input logic rd_en,
    input logic full,
    input logic empty,
    input logic [$clog2(DEPTH):0] count
);
    // 1. No write when FIFO is full
    `ASSERT(no_write_when_full, @(posedge clk) disable iff (!rst_n)  (full && wr_en) |-> ##1 $stable(count) )
```

```
    // 2. No read when FIFO is empty

    `ASSERT(no_read_when_empty, @(posedge clk) disable iff (!rst_n) (empty && rd_en) |-> ##1 $stable(count) )


    // 3. Write increases count (if not full)

    `ASSERT(write_increases_count, @(posedge clk) disable iff (!rst_n)

     (!full && wr_en && !rd_en) |=> (count ==$past(count) + 1))


    // 4. Read decreases count (if not empty)

    `ASSERT(read_decreases_count,@(posedge clk) disable iff (!rst_n)

     (!empty && rd_en && !wr_en) |=> (count == $past(count) - 1) )


    // 5. Simultaneous read and write leaves count unchanged

    `ASSERT(read_write_no_change, @(posedge clk) disable iff (!rst_n)

     (rd_en && wr_en && !full && !empty) |=> (count == $past(count)) )


    // 6. Count must always be in valid range

    `ASSERT(count_range,@(posedge clk) disable iff (!rst_n)  (count <= DEPTH) )


    // 7. Full implies count is DEPTH

    `ASSERT(full_condition, @(posedge clk) disable iff (!rst_n) full |-> (count == DEPTH))


    // 8. Empty implies count is 0

    `ASSERT(empty_condition,@(posedge clk) disable iff (!rst_n) empty |-> (count == 0) )

Endmodule
```

Round Robin Arbiter (2-Requests) Design Code:

```
module rr_arbiter (

    input  logic       clk,
```

```
    input  logic       rst_n,
    input  logic [1:0] req,     // Requests from two masters
    output logic [1:0] grant    // Grant output
);
    logic last_grant; // 0 for req[0], 1 for req[1]
    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            grant      <= 2'b00;
            last_grant <= 1'b0;
        end else begin
            if (req == 2'b00) begin
                grant <= 2'b00;
            end else begin
                // Round-robin arbitration
                if (req[~last_grant]) begin
                    grant <= (last_grant == 1'b0) ? 2'b10 : 2'b01;
                    last_grant <= ~last_grant;
                end else if (req[last_grant]) begin
                    grant <= (last_grant == 1'b0) ? 2'b01 : 2'b10;
                    // last_grant remains unchanged
                end
            end
        end
    end
endmodule
```

Round Robin Arbiter (2-Requests) Assertions Using SVA Macro's :

```
module arbiter_formal (
   input  logic      clk,
   input     logic      rst_n,
   input  logic  [1:0]  req,
   input  logic [1:0]  grant
);

   // 1. Assumptions
   // Stability of requests during a clock cycle
 assume property (@(posedge clk) req == $past(req));

   // 2. Assertions
   // A. Only one grant at a time or none
   property p_one_hot_or_zero(grant);
   (grant == 2'b00 || grant == 2'b01 || grant == 2'b10);
   endproperty
   assert property (@(posedge clk) p_one_hot_or_zero(grant));

   // B. Grant must be subset of request
   assert property (@(posedge clk) (grant & ~req) == 0);

    // C. Fairness – If request is held high continuously,
   // eventually it will be granted (liveness property)
   property p_fairness(i);
   req[i] throughout [*] |-> eventually grant[i];
   endproperty
```

5.4 Benefits and Challenges

The benefits of Assertion – Based Verification are :

Early Bug Detection: Assertions help in identifying bugs at the point of occurrence, making debugging easier and reducing costly late-stage fixes.

Improved Design Quality: By enforcing design intent and expected behaviour, assertions help maintain functional correctness and reliability.

Enhanced Observability: Assertions act as built-in checkers, making it easier to diagnose problems and understand the system's behaviour without extensive debugging.

Reduced Debugging Time: Since assertions provide immediate failure notifications and detailed information on the nature of the issue, they help engineers locate and resolve errors faster.

Reusability and Portability: Assertions can be written in a modular way and reused across different verification environments, testbenches, and projects, leading to a more efficient verification process.

Better Coverage Metrics: Assertions contribute to functional and code coverage analysis by identifying unverified scenarios, improving overall verification completeness.

The Challenges of Assertion Based Verification are :

Steep Learning Curve: Writing effective assertions requires a deep understanding of assertion languages and formal properties, which may be challenging for beginners.

Performance Overhead: Overuse of assertions can slow down simulation performance, especially if too many complex assertions are active simultaneously.

False Positives and Negatives: Poorly written assertions can either fail to detect actual errors (false negatives) or trigger unnecessary warnings (false positives), leading to wasted debugging efforts.

Debugging Complexity: When assertions fail, debugging the root cause can sometimes be challenging, particularly in large and complex designs.

Integration with Legacy Designs: Adding assertions to an existing design may require significant effort in terms of modifying code and ensuring compatibility with older verification methodologies.

Tool and Language Compatibility: Different assertion languages and verification tools may not always be compatible, leading to additional integration and debugging work.

Assertion-Based Verification is a powerful approach for improving design verification by enabling early bug detection, enhancing observability, and reducing debugging efforts. While it requires an initial learning investment, the long-term benefits in terms of design quality and verification efficiency make it a crucial component in modern verification methodologies. To maximize the effectiveness of ABV, careful assertion planning and integration into the verification workflow are essential.

CHAPTER 6

Conclusions and Future Scope

6.1 Conclusion

Formal verification techniques, including Assertion-Based Verification, play a crucial role in modern digital design verification by ensuring correctness, efficiency, and reliability. These techniques provide a systematic approach to verifying complex designs, reducing the dependency on traditional simulation- based methods. The integration of formal methods enhances early bug detection, improves observability, and enables more comprehensive verification coverage. Despite challenges such as a steep learning curve and tool compatibility issues, the benefits of using formal techniques far outweigh the drawbacks, making them an essential component of the verification process.

Formal verification has emerged as a powerful approach for ensuring the correctness of complex hardware and software designs. Unlike traditional simulation and testing methods, formal techniques provide mathematical guarantees about system behaviour, reducing the likelihood of undetected errors. This thesis explored various formal verification methodologies, including model checking, theorem proving, and equivalence checking, applying them to real-world case studies to demonstrate their effectiveness.

The findings indicate that formal verification techniques significantly enhance design reliability by identifying subtle bugs early in the development process. Model checking, in particular, has proven to be effective for verifying finite-state systems by exhaustively exploring all possible states. Theorem proving offers a more flexible but complex approach, allowing verification of system properties at an abstract level. Equivalence checking ensures that optimizations and transformations preserve the intended functionality of a design.

Despite these advantages, formal verification faces challenges such as state-space explosion, complexity in specifying formal properties, and the need for expert knowledge. Addressing these limitations requires advancements in automated tool support, integration with traditional verification methods, and improvements in scalability.

Overall, this thesis contributes to the field by demonstrating how formal techniques can be effectively applied to design verification, providing insights into their strengths and limitations, and proposing methodologies to improve their adoption in industry.

6.2 Future Work

While this research has demonstrated the efficacy of formal verification in design verification, several avenues remain for future exploration:

Scalability Enhancements – The state-space explosion problem remains a major challenge, particularly for large and complex systems. Future research could focus on developing more efficient abstraction techniques, compositional verification methods, and parallel computing approaches to improve scalability.

Integration with Machine Learning – Recent advancements in artificial intelligence and machine learning present opportunities for automating aspects of formal verification. Research into using machine learning models to generate formal specifications, predict verification bottlenecks, or guide state-space exploration could enhance efficiency.

Hybrid Verification Approaches – Combining formal methods with simulation-based testing, fuzz testing, or dynamic analysis could provide a more comprehensive verification framework. Investigating hybrid approaches could help leverage the strengths of both formal and traditional techniques.

Tool Development and Industry Adoption – Many formal verification tools require expertise, limiting their widespread adoption in industry. Future work could explore improving usability, developing domain-specific tools, and integrating formal verification within standard design workflows to encourage broader use.

Formal Verification in Emerging Technologies – As new computing paradigms such as quantum computing, neuromorphic computing, and secure hardware design gain traction, formal verification techniques must evolve to address their unique challenges. Research into adapting existing methods for these domains is crucial.

Automated Specification Generation – One of the significant challenges in formal verification is the manual effort required to write formal specifications. Future research could investigate natural language processing (NLP)-based approaches for translating informal requirements into formal specifications, reducing the barrier to entry for non-experts.

Formal Methods for Security Verification – With increasing concerns about cybersecurity threats, applying formal verification to security-critical systems, including

cryptographic protocols, secure hardware, and embedded systems, is a promising research direction. Developing formal frameworks to verify security properties, such as confidentiality, integrity, and authentication, could significantly improve system robustness.

By addressing these challenges and exploring new directions, formal verification can continue to evolve as a fundamental technique for ensuring correctness, reliability, and security in complex system design.

REFERENCES

[1] "Designing with ARM Cortex-M based System-On-Chips (SoCs) – Part I: The basics," 30 March 2014. [Online]. Available: https://www.embedded.com/designing-with-arm-cortex-m-based system-on- chips- socs-part-i-the-basics/. [Accessed May 2022]

[2] J. G. W. R. Bruce Wile, Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon), Morgan Kaufmann Publishers Inc., 2005.

[3] C. ©. 2. ThesisConcepts, "System On Chip Solution(SOC)," thesis concepts, 2015. [Online]. Available http://thesisconcepts.com/system-on-chip-solutionsoc/?cv=1. [Accessed 9 May 2022].

[4] D. S. a. S. S. Kolay, "SoC: A Real Platform for IP Reuse, IP Infringement, and IPProtection," CAD for Gigascale SoC Design and Verification Solutions, vol. vol.2011, no. Article ID 731957 , p. 10 pages, 2011.

[5] Mohamed, "Improving Reusability in SoC Project Verification Flow," 2019.

[6] Mosensoson, G. (2002). Practical approaches to SoC verification. In Proceedings of DATE User Forum (pp. 05-08). Citeseer.

[7] Stavinov, E. (2010). A practical parallel CRC generation method. Circuit Cellar-The Magazine For Computer Applications, 31(234), 38.

[8] Campobello, G., Patane, G., & Russo, M. (2003). Parallel CRC realization. IEEETransactions on Computers, 52(10), 1312-1319.

[9] C. T. G. Spear, SystemVerilog for Verification, Springer US, 2012.

[10] M. A. B, ASIC/SoC Functional Design Verification: A Comprehensive Guide toTechnologies and Methodologies 978-3-319-59418-7, 3319594184, 978-3-319- 59417-0, Springer International Publishing,
2018, p. 346.

[11] Complex SoC Verification using ARM Processor. [Online].

[12] V. Bertice. Ph.D. Dissertation, Stanford University, August 2003. [Online]. Available: http://web.eecs.umich.edu/~valeria/research/thesis/ thesis2.pdf

[13] https://verificationacademy.com/seminars/2020-functional-verification-study

[14] W. K. Lam. Hardware Design Verification: Simulation and Formal Method Based Approaches,

[15] E. Seligman, T. Schubert, M.V. Acheta Kiran Kumar. Formal Verification: An essential toolkit formodern VLSI design, section Foreword

[16] Kern, C. and Greenstreet, M. R. (1999). Formal verification in hardware design: a survey. Technical report, Univ. of British Columbia, Vancouver.

[17] Bergeron, J. (2003). Writing Testbenches: Functional Verification of HDL Models

[18] Mansouri, N. and Vemuri, R. (2000). Automated correctness condition generation for formal verification of synthesized RTL designs. Formal Methods in Systems Design

[19] A. Gupta. “Formal hardware verification methods: a Survey”. Formal Methods in System Designs,Vol. 1

[20] Lam, W. K. (2009, January 15). Hardware Design Verification: Simulation and Formal Method- Based Approaches.

.

SHINAL_SHARMA_602362033_Thesis_mtech _vlsi_design (2).docx

ORIGINALITY REPORT

15%	10%	7%	8%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	es.scribd.com Internet Source	3%
2	Submitted to California State University, Sacramento Student Paper	2%
3	Submitted to Manipal University Student Paper	1%
4	Submitted to Indian Institute of Technology, Bombay Student Paper	1%
5	"Verification Methodologies", The e Hardware Verification Language, 2004 Publication	1%
6	Erik Seligman, Tom Schubert, M V Achutha Kiran Kumar. "Introduction to systemverilog assertions", Elsevier BV, 2015 Publication	1%
7	Submitted to Maulana Azad National Institute of Technology Bhopal Student Paper	<1%
8	Chong-Min Kyung. "Current status and challenges of SoC verification for embedded systems market", IEEE International [Systems-on-Chip] SOC Conference 2003 Proceedings SOCC-03, 2003 Publication	<1%
9	fliphtml5.com Internet Source	<1%
	www.fcpa.com	

10	Internet Source	<1%
11	Submitted to Institute of Technology, Nirma University Student Paper	<1%
12	research.vit.ac.in Internet Source	<1%
13	www.researchgate.net Internet Source	<1%
14	scholarworks.rit.edu Internet Source	<1%
15	Submitted to Punjab Technical University Student Paper	<1%
16	systemverilog.dev Internet Source	<1%
17	vdocuments.site Internet Source	<1%
18	Harry Foster, Adam Krolnik, David Lacey. "Assertion-Based Design", Springer Science and Business Media LLC, 2005 Publication	<1%
19	www.freepatentsonline.com Internet Source	<1%
20	www.cacnews.org Internet Source	<1%
21	www.ijraset.com Internet Source	<1%
22	Abed, Sa'ed Rasmi H. "The verification of MDG algorithms in the HOL theorem prover", Proquest, 20111003 Publication	<1%
23	Ashok B. Mehta. "ASIC/SoC Functional Design Verification", Springer Science and Business Media LLC, 2018 Publication	<1%

24	programmer.ink Internet Source	<1%
25	hdl.handle.net Internet Source	<1%
26	Submitted to University of Edinburgh Student Paper	<1%
27	repo.poltekkesdepkes-sby.ac.id Internet Source	<1%
28	Introduction to Formal Hardware Verification, 1999. Publication	<1%
29	sdkdocs.cypress.com Internet Source	<1%
30	Submitted to University Politehnica of Bucharest Student Paper	<1%
31	Submitted to University of Technology, Jamaica Student Paper	<1%
32	Submitted to University of Teesside Student Paper	<1%
33	dspace.bits-pilani.ac.in:8080 Internet Source	<1%
34	Safari, Saeed. "System Verilog", Electrical Engineering Handbook, 2006. Publication	<1%
35	Submitted to University of Bristol Student Paper	<1%
36	Submitted to University of Oulu Student Paper	<1%
37	Submitted to Vels University Student Paper	<1%
38	dslab.konkuk.ac.kr Internet Source	<1%

39	eprints.usm.my Internet Source	<1%
40	www.ciol.com Internet Source	<1%
41	Submitted to Universidad TecMilenio Student Paper	<1%
42	V. Zerbe. "New Aspects in HDL's Performance Evaluation", EUROCON 2005 - The International Conference on "Computer as a Tool", 2005 Publication	<1%
43	docplayer.net Internet Source	<1%
44	www.coursehero.com Internet Source	<1%
45	www.grin.com Internet Source	<1%
46	yannherklotz.com Internet Source	<1%
47	Wang, Wen. "Hardware Architectures for Post-Quantum Cryptography.", Yale University, 2021 Publication	<1%
48	fdocuments.in Internet Source	<1%
49	ia903408.us.archive.org Internet Source	<1%
50	ruidera.uclm.es Internet Source	<1%
51	rulrepository.ru.ac.bd Internet Source	<1%
52	Banerjee, Somnath, Tushar Gupta, and Sanjay Gupta. "Logic emulation with forced assertions: A methodology for rapid	<1%

functional verification and debug", Fifth Asia Symposium on Quality Electronic Design (ASQED 2013), 2013.
Publication

---

53 Cristiano Rodrigues. "A case study for Formal Verification of a timing co-processor", 2009 10th Latin American Test Workshop, 03/2009 — <1%
Publication

---

54 Hassan Sohofi, Zainalabedin Navabi. "Assertion-based verification for system-level designs", Fifteenth International Symposium on Quality Electronic Design, 2014 — <1%
Publication

Exclude quotes	Off	Exclude matches	< 8 words
Exclude bibliography	Off		