Verification and Environment Automation for Debug Optimization of Intellectual Property for Safety IP

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

MASTER OF TECHNOLOGY

in VLSI Design

Submitted By

Priyam Jain (602262014)

Under Supervision of

Dr. Ajay Kakkar

Assistant Professor

ELECTRONICS AND COMMUNICATION ENGINEERING
DEPARTMENT
THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
(A DEEMED TO BE UNIVERSITY),
PATIALA, PUNJAB

DECLARATION

I, **Priyam Jain** hereby declare that the work presented in this thesis entitled **"Verification and Environment Automation for Debug Optimization of Intellectual Property for Safety IP"** in partial fulfillment of the requirement for the award of degree of **Master of Technology (VLSI Design)** submitted at Electronics & Communication Engineering Department**,** Thapar Institute of Engineering & Technology (Deemed to be University), Patiala is an authentic record of work carried out under the supervision of **Dr. Ajay Kakkar** (Assistant Professor, Electronics & Communication Engineering Department, Thapar Institute of Engineering & Technology) and under Industrial mentor **Mr. Saransh Mehrotra** (Principal Engineer, STMicroelectronics) from June 2023 to July 2024. The matter presented in this has not been submitted in part or full to any other university or institute for the award of any other degree.

Date: 06 Aug 2024

Priyam Jain
(602262014)

Dr. Ajay Kakkar
Assistant Professor
ECE Department
TIET Patiala

Mr. Saransh Mehrotra
Principal Engineer
STMicroelectronics

CERTIFICATE

STMicroelectronics Private Ltd.
Plot No.1, Knowledge Park-III,
Greater Noida – 201 308.
Uttar Pradesh, India
Tel : +91-120-4003000
Fax :+91-120-4003007
Email : infostm.india@st.com
CIN : U32109DL1990FTC039906
www.st.com

31 May 2024

TO WHOMSOEVER IT MAY CONCERN

This is to certify that **Priyam JAIN** has undergone internship in our **ADG** Group from **Jun 02,2023 to May 31,2024**. He has successfully completed his project on:**IP/SoC Verification.**

During his internship period, he was found to be sincere and professional in his conduct.

We wish him all the best in his future endeavors.

for **STMicroelectronics Pvt. Ltd.**

Parminder Singh WALIA
India Talent Acquisition Shead

ACKNOWLEDGEMENT

The task could not have been completed without acknowledging those who guided and supported me continuously to make efforts successful. Taking this opportunity, I express my deepest gratitude and respect to my supervisor, **Dr. Ajay Kakkar,** Assistant Professor, Department of Electronics and Communication Engineering, Thapar Institute of Engineering and Technology for his guidance and encouragement throughout this thesis.

I express my deep gratitude and thanks to **Mr. Saransh Mehrotra** (Principal Engineer, STMicroelectronics) and **Mr. Abhishek Kapoor** (Staff Engineer, STMicroelectronics) for their valuable advice and suggestions and their unwavering support throughout the thesis. Many thanks to my mentors for their valuable guidance during the thesis, who provided me with valuable advice and support that helped me understand the technical aspects of the thesis. I would also like to thank **Dr. Kulbir Singh,** Head of Department, Electronics & Communication Engineering, Thapar Institute of Engineering & Technology, for giving me the opportunity to work at STMicroelectronics Pvt. Ltd. and providing all kinds of support throughout the thesis. Last but not least, I thank my family, friends, and colleagues for their timely help and valuable suggestions.

Priyam Jain
602262014

Abstract

Verification is the step of aligning a design with the given specifications as per the product requirements. Its primary goal is to ensure that the design of every produced component meets the exact needs of the client. Errors in this process are inherently taken, originating from various modules within the SoC or modules at any time are important to addressing the discrepancies. The IP, as well as its verification process, are presented in this study. The current project on which verification work is done is “Safety IP". This IP works as a source for generating secure pathways and passing them to external ports. The safety IP core module is a unit that controls access to target regions based on configuration access permissions. It interacts with the control unit, a NOC hardware unit that is instantiated within links to interconnected blocks. Taking it as a reference, environment setup is done for AXI/APB protocols using python scripts.

As designs become more complex and scale down to the nanometer scale, the problem of debugging becomes more difficult. When the simulation regression is complete, it should fix the reported failures. The process of analyzing and debugging these failures is manual and used more resources. This approach helps automatically classify failures based on their characteristics. Itthen automated the triangulation which determines if failures are in the design or test bench. It uses the dataset to identify errors and divide them into different bins. It then uses these bins to train models to detect and identify errors due to actions that do not complement the original and to trace the root cause of failure in a particular container. The learning model is based on CNN with several abstraction layers and tools used are Cadence Xcelium, Vmanager. It gets the debugging coverage result of 91.9%. Aspects covered are register test cases and repetitive checkers functional test cases.

LIST OF CONTENTS

S. No.	Title	Page Number
1	Declaration	ii
2	Certificate	iii
3	Acknowledgement	iv
4	Abstract	v
5	List of Figures	ix
6	List of Tables	xi
7	List of Abbreviations	xii
1	**Introduction**	**1 - 6**
1.1	Intellectual Property	2
1.2	Verification and its Levels	2
1.3	Types of IP Level Verification	2
1.4	Verification Cycle	3
1.5	Verification Challenges	4
1.6	Design Verification Approaches	5
2	**Literature Survey**	**7 – 17**
2.1	Problem Formulation	13
2.2	Challenges	14
2.3	Analysis	16

2.4	Objective	17
3	**Proposed Methodology**	**18 - 38**
3.1	Methodology	18
3.2	Proposed Work	20
3.3	Flowchart	22
3.4	Verification Methodology	23
3.4.1	SV UVM based Methodology	23
3.4.2	Assertion based Methodology	24
3.5	IP Interface	24
3.5.1	APB Interface	24
3.6	UVM Component	30
3.6.1	UVM Sequence Items	31
3.6.2	UVM Sequencer	32
3.6.3	UVM Driver	32
3.6.4	UVM Monitor	33
3.6.5	UVM Agent	33
3.6.6	UVM Scoreboard	34
3.6.7	UVM Environment	34
3.6.8	UVM Test	35
3.6.9	UVM Top	35
3.7	Safety IP	35

3.7.1	Key Features	36
3.7.2	Configuring the Safety IP	37
3.7.3	Environment Setup for IP	37
4	**Results and Discussion**	**39 – 50**
4.1	Comparison Results	45
4.2	IP Verification Result	47
5	**Conclusion and Future Scope**	**51**
6	**References**	**52 – 55**

LIST OF FIGURES

Fig. 1.1	Design Cycle	1
Fig. 1.2	Usability of IP in SOC	2
Fig. 1.3	Verification Cycle	3
Fig. 1.4	IP Verification Flow	4
Fig. 1.5	Debugging Approach	5
Fig. 3.1	Working of ML Model	19
Fig. 3.2	Verification Environment Setup	20
Fig. 3.3	AI Model Generation Flow	21
Fig. 3.4	Flowchart	22
Fig. 3.5	UVM based Methodology	23
Fig. 3.6	APB to AHB Bridge Flow	25
Fig. 3.7	APB State Diagram	26
Fig. 3.8	Different ATB Signal	27
Fig. 3.9	Waveform Analysis	28
Fig. 3.10	Waveform Analysis State	28
Fig. 3.11	RAL Model	30
Fig. 3.12	Sequence	31
Fig. 3.13	Sequencer	32

Fig. 3.14	Driver	32
Fig. 3.15	Monitor	33
Fig. 3.16	Agent	33
Fig. 3.17	Scoreboard	34
Fig. 3.18	Environment	34
Fig. 3.19	Top Module	35
Fig. 3.20	Safety IP top level	36
Fig. 4.1	Convolutional Layer Generation	39
Fig. 4.2	Model Training	41
Fig. 4.3	Model Accuracy Result	42
Fig. 4.4	Bins Accuracy Compression	43
Fig. 4.5	Model Accuracy vs Wn Values	44
Fig. 4.6	Bins Failures Output	44
Fig. 4.7	Distribution Samples	45
Fig. 4.8	Waveform Analysis	47
Fig. 4.9	Coverage Analysis	48
Fig. 4.10	Regression Analysis	49

LIST OF TABLES

Table 4.1	Bins Accuracy Result	42
Table 4.2	Result Comparison with existing approaches	46

LIST OF ABBREVIATIONS

IP: Intellectual Property

CPU: Central Processing Unit

SoC: System on Chip

IC: Integrated Circuit

DV: Design Verification

RTL: Register Transfer Logic

HDL: Hardware Description Language

AMBA: Advanced Microprocessor Bus Architecture

SV: System Verilog

OOPS: Object Oriented Programming

DUT: Design under Test

UVM: Universal Verification Methodology

ADAS: Advanced Driver Assistance System

ECU: Electronic Controller Units

TCM: Tightly Coupled Memory

FIFO: First In First Out

RAL: Register Abstraction Layer

VIP: Verification IP

ML: Machine Learning

XML: Extensible Markup Language

NOC: Network on Chip

AI: Artificial Intelligence

ReLU: Rectified Linear Unit

Chapter 1

INTRODUCTION

Verification involves the process of meticulously checking and validating the functional accuracy of a design against its given specifications. Its primary objective is to ascertain the operational integrity of the proposed design [13].

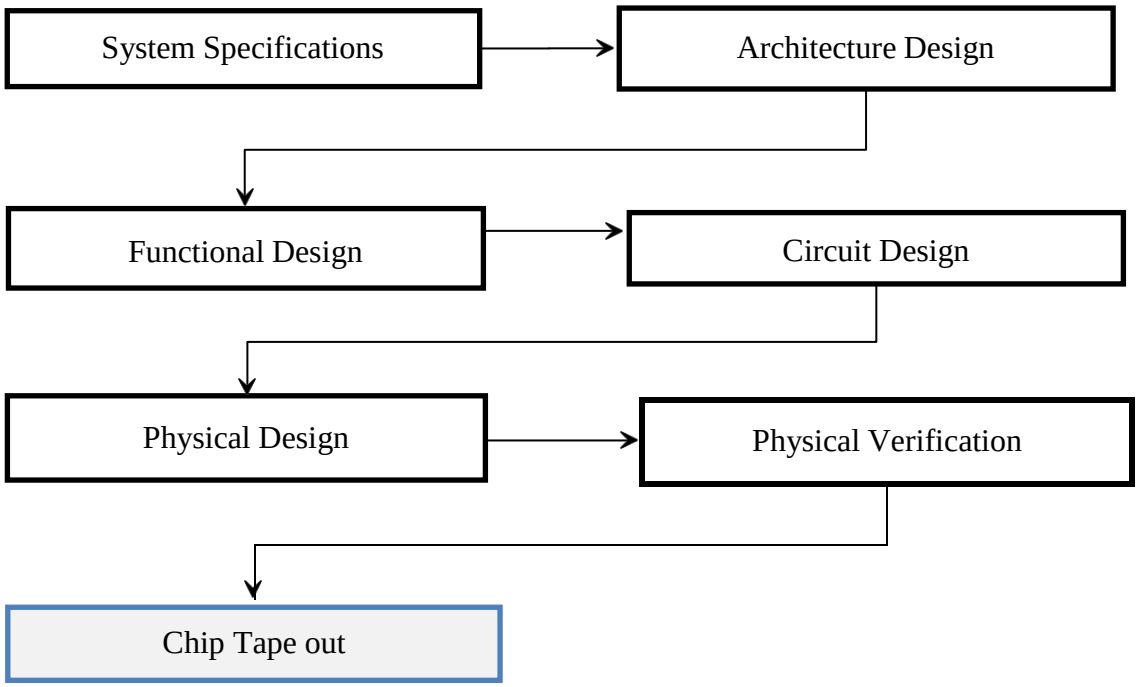


Fig. 1.1 Design Cycle

Fig 1.1 illustrates the key stages of the design cycle. Initially, the design cycle includes defining system specifications, establishing architecture, detailed specifications and simulating functionalities, creating circuit schematics, designing the physical layout, and verifying the design to check for the specifications as per the requirements [3, 24]. Functional verification validates the operation of the design. At this stage, the RTL and verification teams collaborate with each other as a design and verification engineer to ensure the validity of the RTL code of design and work on assertions and coverage metrics for the process towards the final chip tape out [5].

1.1 Intellectual Property (IP)

IP serves as the main component of SoC development. IC design works by consolidates numerous specific functions as per the block guide into a single chip, where pre-designed IP is crucial to enhance architectural complexity [5,42]. This is facilitated by the adoption of standard CPUand functionalities of particular IP as an interface like switch, clock, reset across the SoC which are passing through NOC so that these IPs can be efficiently reused.

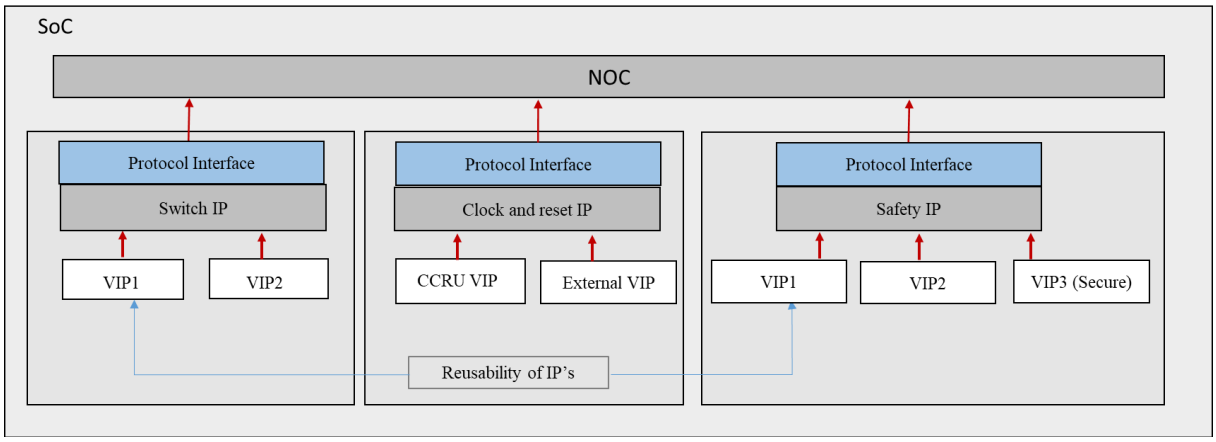


Fig. 1.2: Usability of IP in SoC

1.2 Verification and levels of Verification

The process of verification standards presents numerous verification challenges that can be effectively addressed through advanced methodologies and optimizing reusability [7]. Verification occurs at multiple levels:

i. IP Level: The primary goal for the IP verification teams is to validate each specifications and user blocks of the IP and its functionalities to ensure the accuracy of its design [30].

ii. Subsystem Level: At this stage, the focus is on verifying all essential subsystems like connectives among blocks and modules before their integration into the main SoC [24].

iii. SoC Level: Peripheral IPs and other non-critical components are integrated into a single RTL code to make proper connections in the chip, to be verified by SoC verification team [38].

1.3 Types of IP Level Verification

i. Direct: This verification process involves understanding the IP block guide and specification list and developing targeted test cases based on the verification plan to verify the IP's functionality. [14,20].

ii. Constraint Random: This process involves utilizing randomly generated seed values from the various constraints used simulator to conduct tests. Each test iteration sets various variables according to the seed value, ensuring different values are tested each time [39].

iii. Formal: In this verification approach, the tools automatically generate input stimuli, and the properties are verified by written SV assertions [8]. Debugging these assertions can be challenging,especially in complex architecture. Functional based verification only evaluates the design using a limited set of test cases but formal verification deeply examines all possible states assertions of the architecture [16,12]. This exhaustive analysis ensures that the design meets the specifications under every possible condition.

1.4 Verification Cycle

The semiconductor verification cycle involves planning for comprehensive testing coverage, multiple debugging to ensure correct operation detailing specifications, implementing test simulations, multiple times debugging to ensure correct operation, validating behaviors with assertions,process of documenting methodologies and results, and achieving verification closure through writing formal assertions and functional test cases before advancing in development [3,29].

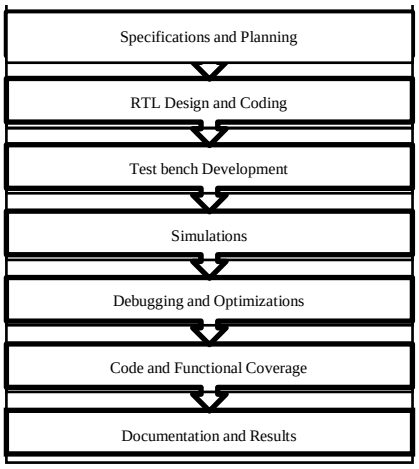


Fig. 1.3: VLSI Verification Cycle

1.5 Verification Challenges

Engineers face increasing workload to deliver complex projects quickly, this is due to the rising complexity and high client product demands. Key challenges include:

i. Execution: Managing execution of projects efficiently using block level design techniques to boost productivity for both design and validation teams [25].

ii. Efficiency: Decrease the manual intervention while doing design verification to enhance efficiency and accuracy [24].

iii. Reuse: Increase the reuse of verification objects and components across new projects through defined structures and its effective documentation [40].

iv. Comprehensiveness: Ensuring verification covers all design functionalities by optimizing productivity and reuse efforts which is supported by the specialized verification team and standard tools [33].

These challenges occur due to the complex verification flow after getting specifications from the client and then RTL is designed as per the specifications, which is then functionally verified by the verification team [11,34]. Synthesis is done for placement of blocks and then adjust these blocks as per the internal structure known as routing after which tape out process in fabrication units needs to be done.

Fig. 1.4: IP Verification Flow

1.6 Design Verification Approaches

For years, studies have shown that the percentage of time and resources devoted to verification has increased for each new chip generation. Thus, overall, verification is growing faster than other stages of semiconductor chip development and client projects [38].

This chips industry faces a critical challenge in managing the increasing design complexity caused by growing process flow and reducing chip manufacturing time.Traditional methods for these tasks are uses more resources [2]. In spite of this, some data based trained model methodologies as AI provide automated solutions for handling integration of design and its testing. Automated error debugging techniques reduces the efforts needed at the abstraction level, leading to improved chip manufacturing productivity [29,33].

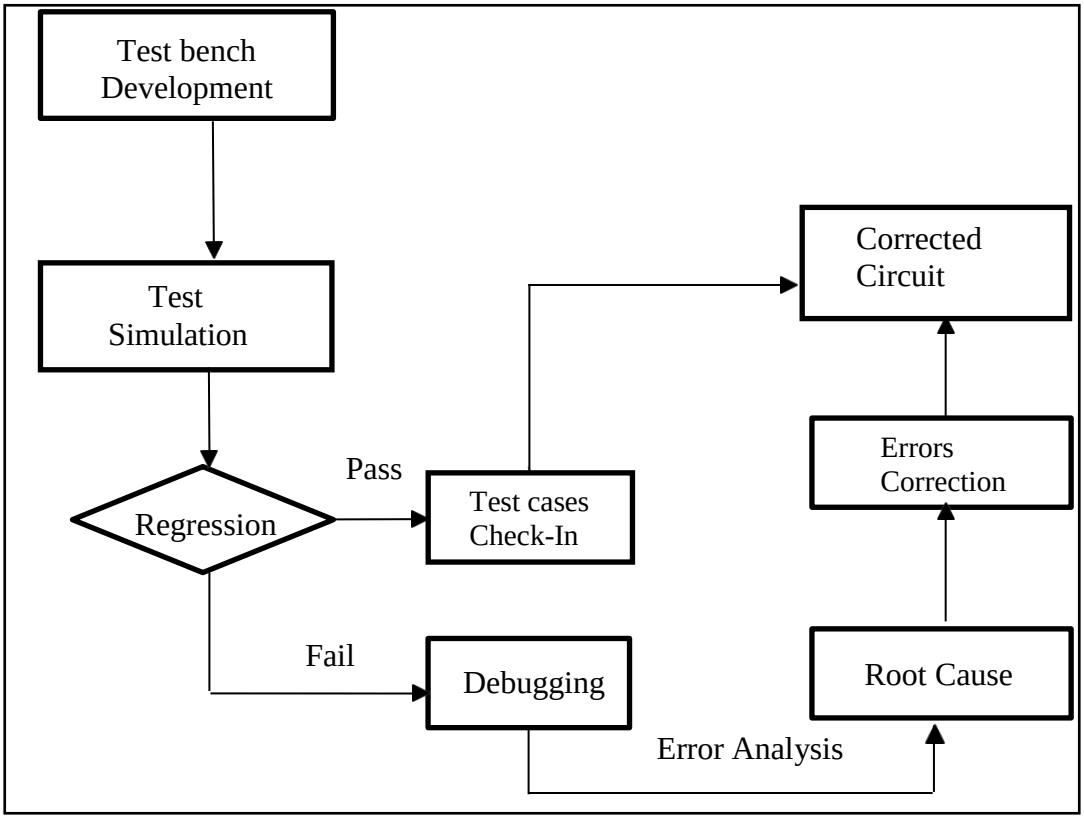


Fig. 1.5: Debugging Approach

Studies suggest that the amount of time and resources devoted to testing increases with each new chip, as chip development and projects progress more rapidly [16]. A major challenge for the manufacturing industry is developing ways to reduce design complexity due to evolving flow changes [25,35]. AI provides an automated approach to overcome this manual way to debug and find the root cause of originating bugs [17]. Initially, the test bench is developed for simulation purposes after which regression runs. If the test case fails, then debugging is required to find its root cause and then correct the error for new regression else test cases check-in to the regression database for the corrected circuit.

To better understand how this approach helps, consider a chip inspection flow. The architecture team builds chip models to analyze system performance and RTL models are developed by the design team to fetch the coding errors [3,40]. The error-checking process is initiated at the same time, the test bench is designed, and tests are simulated according to the test plan. The simulation results are corrected, and the regression is rerun until the validation goal is achieved.

This debugging system automatically detects and finds the root cause of the regressionerror by using AI. It identifies and analyze raw regression and the root causes of design and testing platform errors by identifying the relationships among the simulation log failures based on their features [6,15]. Extraction includes analyzing and clustering simulation log failures of the same issue type into the same groups, and waive rules enable users to exclude specific error messages that are deemed insignificant or vice versa using this model [21]. Create error solutions or root cause tags to identify compilation errors and possible solutions for the same. This pre-trained model generates a report that consists of an analysis for all the failure clusters and then sends the report to the user.

Chapter 2

Literature Survey

Contemporary designs incorporate a complex ecosystem of owner and third party developed IP design blocks. These SoC integrate protocols, specifications based specific functions and contributions from multiple teams, resulting in a system that is highly intricate and challenging to verify and debug. To debug these failures there are some existing approaches, this chapter introduces the research work done by various scholars in the functional and formal design verification and automation.

A. Rezaei Aderiani *et al.* [1] aimed to optimize the geometrical quality of verification assemblies by simulating and adjusting the design corner cases address locations using the concept of twinning process. Not able to relocate addresses to optimize for the formal assertions.

A. Wiemann [2] proposed standardized approaches in formal and functional verification of RTL. Worked on generating standard checkers unit and corner cases to make it more reusable in IP verification. It worked only for the corner scenarios test cases.

Ahmed Wahba *et al.* [3] proposed a method for handling growing complexity of digital circuits for debugging failures using context based failure root cause. Hence automating any stage specifically at the stage of finding the root cause has a significant effect on reducing the manufacturing time.

Ali *et al.* [4] proposed a random verification strategy using defined constraints from bi directional set of expressions that enables the scenario-based testing to validate diverse operational scenarios of the chip, thereby enhancing the thoroughness of the verification process.

Amr Hany *et al.*[5] introduced an approach for functional verification of the design flow within designed RTL by generating valid coverage assertions properties to narrow down the

efficiency and reliability of designs.

Ankitha and D. Aradhya [6] identified the application of scripting in design verification for the automation technology integrating SV with the help of Cocotb application. The focus was to make effective design simulations and test cases.

Arpitha O. *et al.*[7] optimized integration of multiple blocks of IP's in a SoC and to ensure the functionality of IP cores and connection of interconnects using NOC based signal topologies within complex digital architecture.

Ashima Kukkar *et al.* [8] conducted a Deep learning based classification methodology for failures of a SoC architecture for preprocessing and duplicate bugs modules. Extract relevant features which was required to debug from these modules and give analysis but only for the duplicate bugs.

Ashok B. Mehta [9] proposed constrained random validation techniques for verifying the digital IC. The focus was on generating the effective randomization through multiple constraints that enhance reliability and functionality in the specifications provided.

Binod Kumar *et al.*[10] identified post-silicon error detection through strategic selection of signal line to get the enhanced system reliability. It has the limitation that it worked only for the protocols signals with limited internal signals observability.

Brian Keng and Andreas Veneri [11] introduced algorithms for design debugging that use directed process to analyze the errors in an environment framework, effectively managing excessive errors. It reduced the read only memory region of debugging while checking the incomplete results of recent used memory.

D. Craigen *et al.*[12] analyzed formal methods and their implementation in the verification of the design. Used formal properties of assertions methods generated from the object assignment in the script and also stated the ways to make it more reusable.

Daeseo Cha *et al.*[13] introduced automated test bench generation technique using metadata of design specifications created from chip-level structural test bench for verification of registers and inter connectivity among the multiple blocks.

Djordje Maksimovic *et al.* [14] identified regression with many similar root cause in particular run, and to rank them and expedite debugging. Utilized classifier ML technique along with historical data set result of the functional debugging. Aimed to rank regression test cases based on their analysis of log based regression results.

E. Mankolli and V. Guliashki [15] introduced an aggregation technique for object detection and recognition through the unsupervised learning with an auto language processing technique. This electronic design automation framework reduced the design effort for RTL verification in digital circuits while also considering performance.

El-Ashry and S. Adel [16] emphasized the RAL as a critical component of the verification environment, provided access to registers via frontdoor and backdoor paths which creates reference model. This reference model within this environment plays a pivotal role in validating training data consistency and overall functionality.

Eman El Mandouh and Amr G. Wassal [17] introduced deep model training, which permitted to bypass objective functions, basically a strategy for handlingcombinatorial problems. Aimed to detect defects by using K-means clustering algorithm with fault injecting into the netlist and compared the result until it matches the defined clusters.

Eman El Mandouh *et al.* [18] worked on debugging framework that provide regression based clustering and visualization techniques to navigate towards failure. They are designed to primarily handle RTL debugging issues but drastically limit their ability to handle complex functional debugging efficiently as for the SoC test cases.

Ganapathy Parthasarathy and Aabid Rushdi [19] introduced an approach from running regression to get the extracted failures from the complete set of clustered test cases. To estimate

the possibility that a test will carry bug or not, model learns the features of both RTL code and tests bench which are introduced by the design modification.

Gaurav Sharma *et al.* [20] proposed a framework to automate the validation of complex IP with access to different fields. The RAL model frontdoor values justifies the coverage of multiple blocks when compared to backdoor values. This finds and validated bugs only for the register test cases.

Gopal Sharma *et al.* [21] derived a test intent on a UVM RAL environment to generate automated test cases. It improved functional coverage metric through a bug expose algorithm. It saves the manual effort of writing test cases and improves the functional coverage closure in less number of transactions.

Goldie A. Smith *et al.* [22] proposed a method of Boolean satisfiability for diagnosis the root cause of bugs through systematic logical evaluations of defined contexts, provided efficient address localization and automated capabilities in digital systems.

Harshita B. *et al.* [23] proposed test case scenarios developed for test bench extracted for each features of the controller unit of environment, automating their execution using UVM within an electronic design automation using cadence tool.

Hussien, K.Bier *et al.* [24] presented a UVM verification method centered on a comprehensive verification environment comprising of agents. Each agent represents a specific data protocol connected to the DUT through interfaces, facilitating the communication within the verification environment. It did not supports external modules and supports only limited built-in macros.

J. R. Goldberg [25] studied the verification and validation work in the register access methods for the UVM environment and created standard register access fields and bits reassignment in these registers fields. It was limited only for register test cases.

Joon-Sung Yang and Nur A. Touba [26] presented an approach to select signals to be observed for pre detection of digital design issues. Determined the propagation in combinational circuits by bypass the signals created possible errors but not able to find the root cause of a bug.

Liang K. Wu *et al.* [27] designed an associative memory controller and conducted functional simulations for coverages using cadence simulator. While the simulations captured typical functional behaviors, it was found that comprehensive code coverage statistics were not reported.

Mustafa Efendioglu *et al.* [28] aimed to localize complex system C model errors. Extract features from the model for checkers and properties of large SoC designs to debug its illegal access among design. This model identifies a list of signals responsible for the illegal violation.

Priyanshi Gaur *et al.* [29] proposed a ML based solution for failures using multiple constraint randomization. It involved ML model training using clustering method to predict the probability of the output even when it had little information of digital set of the architecture. It had limitation that the model trained with limited sets of dataset.

R. Cheng *et al.*[30] proposed an efficient gradient technique used for standard cell placement, effectively integrating this technique into the module pipeline. This was to update the earliest design debugging methods which were based on the related fault diagnosis techniques. It has limitation that it only finds faults localization of bugs.

R. W. Brennan et al. [31] proposed the process of integration of components and objects to enhance embedded verification. Focuses on methodologies that are aimed to improve the exiting responsiveness of the automotive SoC electronics design.

S. Safarpour et al.[32] aimed to enhance the accuracy of observable Boolean functions for the digital circuits through the concept of satisfiability of undefined scopes in the test cases. This approach not able to worked for combinational circuits having multiple vectors test cases.

Samhita Varambally B. [33] proposed a ML method to improve bug-catching efficiency and verification by doing the assertions coverage. It used supervised learning with artificial neural networks on an open-source platform to detect the bugs. It was observed that it finds bugs only at the core level.

Sebastian Siegfried Prebake *et al.* [34] identified challenges in doing formal and functional debugging due to its complexity. Worked to handle these functional debugging using a graph based model transformations but not able to finds architectural bugs.

Sherif Honsy [35] proposed a method where the reference model generated using standard test bench critically verifies the logic of all the blocks connected with the firmware driver by cross-referencing data with the expected results of scoreboard during the random test phase.

Srinivasan A. [36] identified historical progress of ML and its impact through new innovations on future developments. Stated that the evolution of deep learning methods required a random-decision based system that integrated into the design systems for effective verification. Validation is done by using ML cocotb technique in design verification through scripting language.

Varghese M P *et al.*[37] introduced a ML methodology that increased the layers for circuit's simulations based on neural network models and provided the techniques to improve results for working on datasets. This approach made it challenging to develop classifiers with imbalances found in training model.

Wang *et al.* [38] highlighted UVM used in simulation verification, leveraging features such as the sequencer processed items for reusable stimulus generation, objection process for the simulation control. These features contribute significantly to the efficiency of verification across digital circuit designs.

Yugeshwari Nemade and Prathmesh Pawar [39] developed an automated test bench that reduced

the time required to design a test bench. It was made for the APB architecture. Generated Test bench is compared with the test bench generated by the Quest sim, where Quest sim generated test bench gives better accuracy.

Yugeshwari Nemade [40] proposed a UVM based design verification test bench for the APB bus architecture which was created using scripting language. For validation purposes, the same test bench was only generated using the system software. The scripting created template was compared with the software created test bench, resulting in the scripting created template being more efficient.

Yu-Shen Yang *et al.*[41] aimed to automate debug properties along with their bugs that emerged due to trace buffered coding scenarios RTL by introducing an automated setup to show how these bugs are different from non-traceable bugs.

Zhou *et al.* [42] introduced a test bench environment that omits an object from referenced component. This test bench targets core level verification, specifically focusing on functionalities of particular core within each block, without incorporating formal properties and methodologies.

2.1 Motivation and Problem Formulation:

The differences involved in the adjustment process makes it difficult, and the results are often unpredictable. Initially, a test run against the RTL to test whether the model behaves as expected as a bug that needs to be resolved. If the error is in the design, sometimes it is needed to run a few simulation cycles else changes in the test bench are required. This is often helpful when running simulations for millions of cycles to find strong corner pockets embedded in the design.

The size of the chip is a factor, but the performance of the chip has more influence on the debugging process. The design requires thousands of parallel processes to simulate a certain situation. To identify the faulty branch, one needs to inspect parallel events. The complexity is

proportional to its size. The final implementation of the chip is also necessary. Consider debugging C code and RTL in processor design. The debugger must be smart enough to understand how the gate or transistor works to measure each impedance.

The variances involved in the debugging process make it painstaking, and the results are often unpredictable. Initially, run the stimulus against RTL to verify that the model behaves as expected or tagged as an error to be debugged. If an error is determined in the test bench, then rewrite this else check-in and raise the defects against design team. Most of the time, error is in the design, hence running multiple simulation cycles isvery important. Often, it would help to debug multiple test cases which are failing for the same root cause.

In this, discussion is about what makes debugging so tricky and what features of a debug tool can help ease the process. Driven by application areas like ML, chips are growing more extensive, making the debugging effort much complicated. Chip size is one factor, but chip functionality is even more impactful on the debug process. A design needs thousands of transactions in parallel to simulate a specific condition. To isolate the bug, one must check parallel events to find which branch is the culprit. Today, it is asmuch about the exposed bug's complexity as its size.

Consider debugging C compile code with RTL in processor design. In power designs, it is recommended to add separate lines or other energy saving devices that are not used during the design process. Debugging tool is able to understand the operation of gates or transistors for reducing the gauge errors. Detailed debugging is especially important for applications where safety compliance is essential, such as automobiles.

2.2 Challenges:

To clarify the above mentioned issues in section 2.1, there are some significant issues facing by the design and verification team while designing core modules. The fundamental problems of bug detection of faults in design elements is the best practice for delivering quality designs.

Complexity issues reduced by these initial design verification before doing exact modules design approached to reduce architecture errors. Maintaining good architecture creates challenges later. This means that efficiency is hard to achieve and gets far away from RTL code, with the risk of complexity, associated errors and cost. Today, a modern and efficient method is used to create a virtual prototype that runs actual software and exact I/O data.

At this level of abstraction, it is very important to explore real state decisions and analyze the possibilities and strength of the proposed approach. As hardware designers begin to generate RTL code during the development cycle, it is helpful to use the program to ensure that the RTL is designed right. Over the past two decades, constrained stochastic methods such as UVM have become essential in RTL simulation, making these methods helpful in designing and implementing diagnostic tools.

Computer hardware and simulation engines have improved efficiency and decision-making capabilities over the past decade. At the same time, legal writing functions and validation models have become as complete and efficient as ever with error-hunting techniques. Suppose dynamic simulation and static data are needed to solve a problem, in that case, it is required to quickly move our problem to a dynamic platform, which allows actual software to run on a few megabytes of system or chip level. Mistakes at this stage will be more complicated, if any of these remain then they will cause problems.

Studies have shown that debugging is more complicated while doing verification process of IP designs. In general, designers are involved in correction, this usually involves some level of waveform and code analysis. Engineers are most excited for the process of fixing and testing errors. The testing team needs availability and consistency of debugging features and experience with different set of test platforms as simulation acceleration. The infinite search of debugging root cause should be measurable, which can only be competed through proper planning and execution of planned test cases.

Work or scope requires the complete concept plan based on engineering knowledge and design

knowledge along with the effective planning and quality control standards. Evidence is an important task to find and doing analysis of test errors, as the team must organize, analyze, and imagine multiple sources to build relationships and agree on the necessary approval and validation processes.

2.3 Analysis:

Improving accuracy, speed, and overall debugging is a critical challenge. Therefore, the time it takes to get an answer can vary greatly depending on the localization of the error and the line of code involved. Many new architectures are specific to the registers that must be able to understand the transistor methods to perform proper testing to debug the device. Evidence based verification methods help to cross-check target behavior and have the advantage of improving stability and low-level performance to catch more errors faster and more efficiently. Supports bug hunting at multiple levels of the hardware development lifecycle, reduces the workload of verification engineers. By this analysis it can be stated that there is a requirement for an automatic debugging system. At the other end of the development lifecycle, the way is to carry out power analysis and to fix power bugs while running actual hardware and also maintaining high-performance.

To complete transition between simulations and prototyping environments, integration of bug analysis with several automated debug platforms is required, which allow us to debug bugs across all scenarios and at different levels to increase debug efficiency. For bug absence during tape out stage, complete test plan and coverage analysis of code and branches are done which allowing one to plan, execute and sign off after getting coverage goals. Therefore, the challenges of bug reduction when detecting and debugging errors in complex hardware are apparent to detect errors efficiently and effectively. Hence there is a requirement of a debugging technique which debug the bugs and also the possible errors root of cause and also to enable the bug predictions based on the previous regression results. This is for quick and effective debugging to do the analysis when encountering bugs. Although these challenges getting worse when the size of cores and modules increases, hence verification tools and advanced techniques are needed to meet client product demands.

2.4 Objectives:

From analysis, the following objectives have been drawn:

i. To review and analyze ML method to automatically identify and classify design/test bench bugs.

ii. To develop a ML model to streamline the debugging process for optimization overall chip development productivity.

iii. To improve the accuracy for manual bugs analysis by automating the repetitive tasks or debugging related tasks.

Chapter 3

Proposed Methodology

After reviewing the various approaches present in literature, there is a requirement to develop the model which has a dataset with an effective training phase for the accurate prediction of bugs. Proposed scheme saves the time and efforts to manually find the failures of the same root cause for the case of IP and SoC verification.

3.1 Methodology:

Employing automation for debugging errors is required at the abstraction levels. It is well understood that verification and debugging take much time and are the most challenging parts of chip development. Every time a regression fails, the verification team must examine the reports and debug the causes of the failures. Simulation regressions are run countless times during a project. Because RTL and test bench code constantly changes as bugs are fixed and new features are added and tested, recurring bugs occur daily. Manually handling log regression errors is a heavy burden for the teams therefore introduction of debugging system is required which automatically detects and finds the root cause of the regression error. It also uses AI to identify and analyze raw regression results and then find the root causes of design and testing platform errors.

AI has proven helpful ways for many applications, providing intelligence and speed to see insights from multiple devices. Another helpful feature of the debugging tool is the ability to reuse test bench setups in different environments, thus speeding up time. Now with this efficient debugging and integration technique, time and can be saved and hence debugging is easily implemented for all design and analysis processes. It extracted patterns and with these patterns, dataset is generated which typically reduces debugging time to focus on more productive tasks. AI helps find the root cause of design failures as being able to identify the exact fault is very difficult because there are so many clues that need to be identified and understood. It would help if anyone run multiple simulations and tests to validate the design without a debugging tool with all the necessary capabilities. It is divided into two mode phases,

which are the training phase and the testing phase. In the training phase, dataset generated from extracted patterns to train models to predict values and in testing phase logs are extracted and validate the bins and compare with the actual values. In testing phase unknown patterns are extracting log errors and validated the respective bins to find out the model accuracy. This means that all the data that must be analyzed has zero errors.

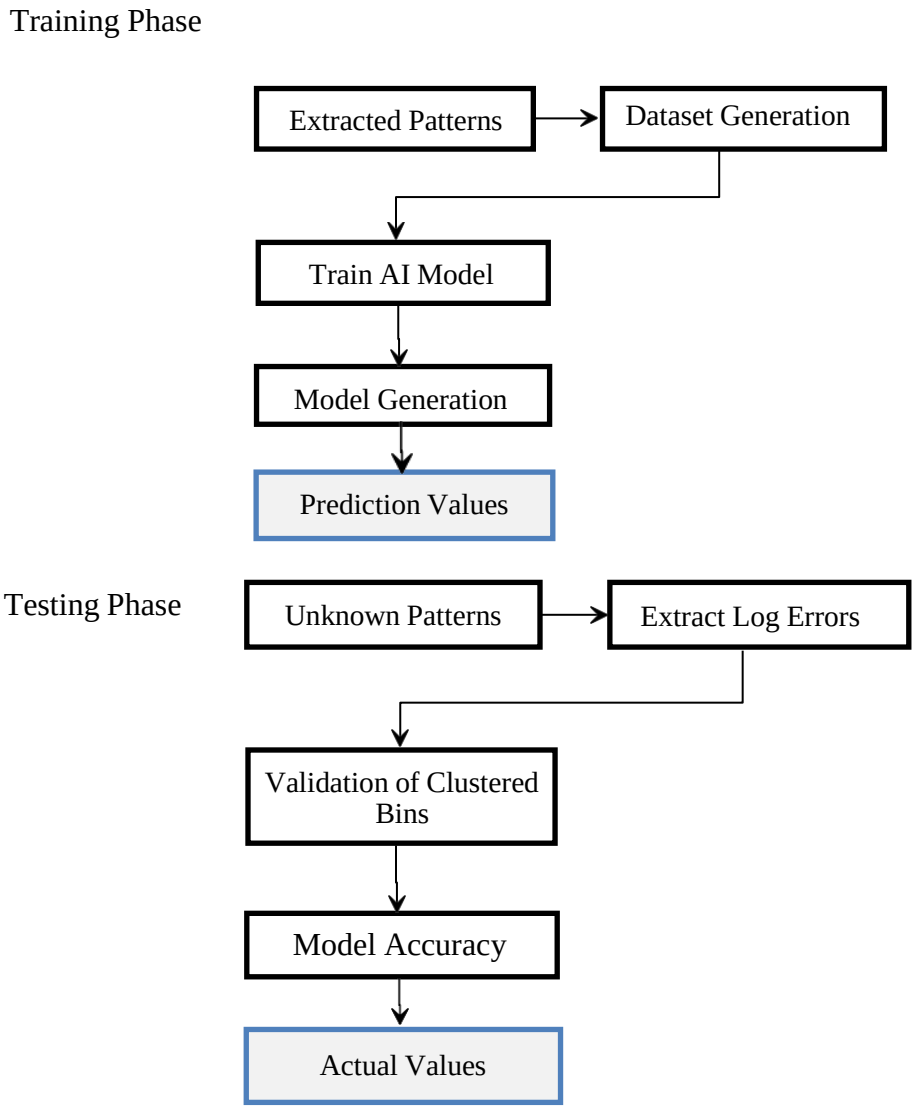


Fig. 3.1 Working of ML Model

Initially, test cases are checked in the database by using commands to launch the regression, it gets results of test cases where failing test cases are only supposed to get dealt with. All the failing test cases have failure logs, initially open these log files by giving directories paths in the configuration file. Using these log files, features such as errors, user defined patterns or keywords are extracted to deal with the model. Based on this dataset, make use of the ML k-means clustering algorithm to list the similarities and differences of failures or other inputs the user provides. This is clustered into multiple bins, these are used to analyze results for the following regression and to train a model to predict the subsequent failures.

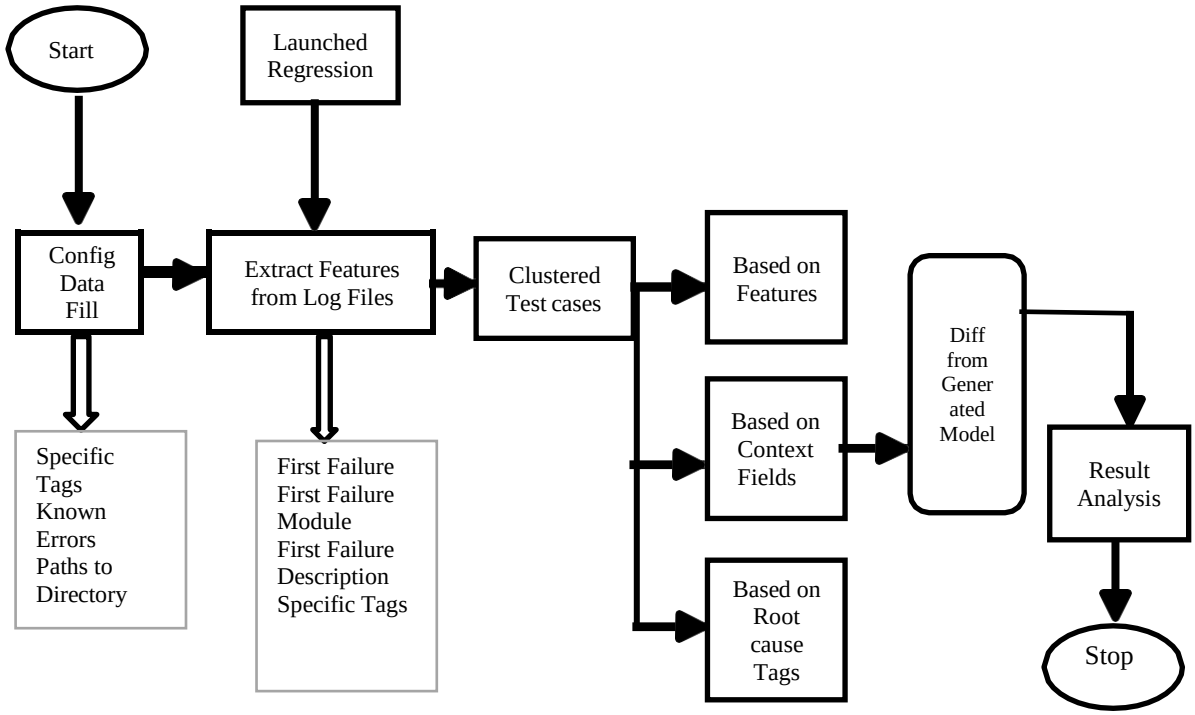


Fig. 3.2 Verification Environment Setup

3.2 Proposed Work:

The last step of the verification process is debugging, which is very close to simulation. The warning message must be debugged once the simulated regression run is complete. New bugs are constantly created as code evolves to fix bugs or add functionality. This method divides the failures into multiple error boxes based on error characteristics, which will help debugging

easier. These errors are considered to determine whether the error is from the design or the test bench by finding the failures context.

This method is used to troubleshoot the faults. It uses the dataset to identify errors and divide them into different bins. It then uses these bins to train models to detect and identify errors. While reducing time to debug, it is also recommended to avoid getting debug failure which is required to understand the cause and to fix the errors. This framework optimizes the regression errors analysis for debugging, which involves collecting and analyzing messages from the simulation log file. Initially, running regression is required after the standalone test case gets completed, it generates the report on the tool, which is exported as a CSV file. This CSV file is used to extract the required data to generate a dataset for training our AI model. Data is collected with multiple columns with information such as test case names, sequences, error types, and first errors.

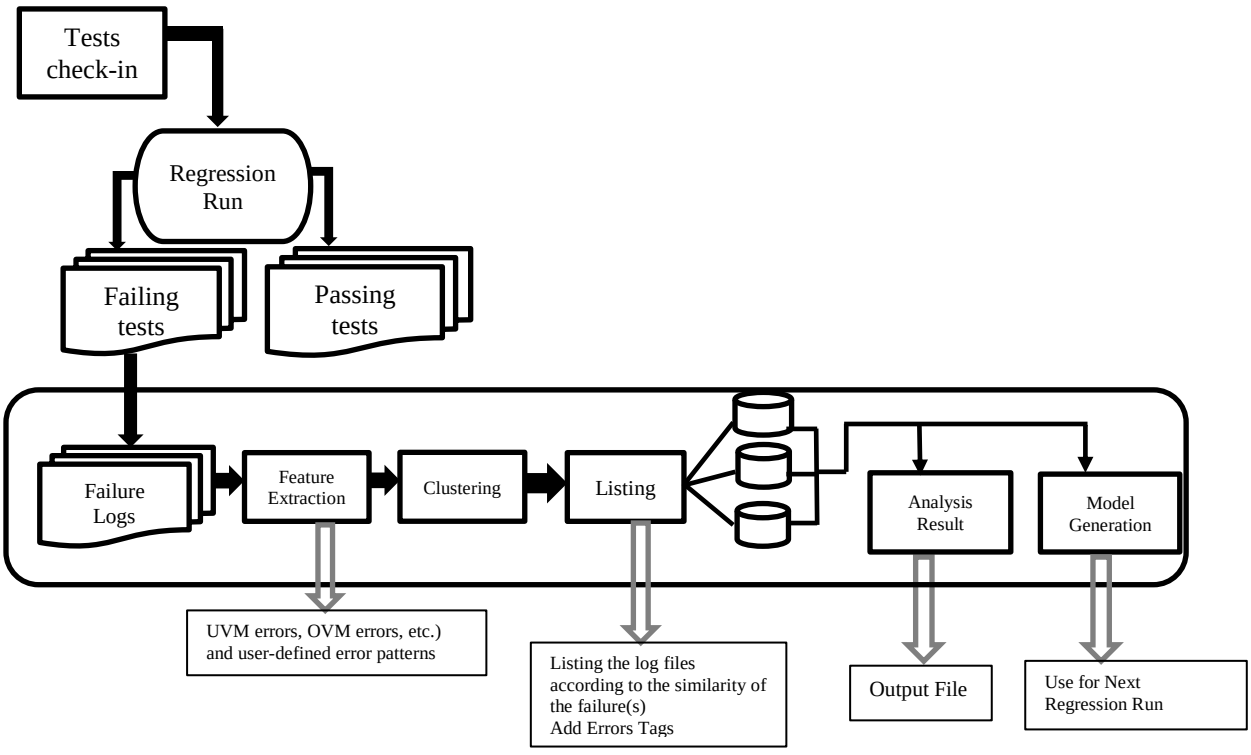


Fig. 3.3: AI model generation flow

Identify relationships among the simulation log failures based on their feature extraction. It also added the feature of waive specific contexts which enables users to exclude specific error messages. Create error solutions/root cause tags to identify compilation errors and its possible solutions. Initially, the user must fill a configuration file for path directories, specific tags, and known errors. After regression launched, the failing tests have generated the log files. From these log files feature extraction is done as module names, failure descriptions and specific tags need to be covered. This extraction is done by using export data from the vmanager tool. After extraction, clustering is done using ML algorithms on the tool, by writing a python script. It will generate the multiple clustered test cases bins based on root cause tags, context fields and listed features. It will also classify whether the error is from design or test bench. It will match with the already listed defect, then do the functionality checks by taking differences from the results.

3.3 Flowchart:

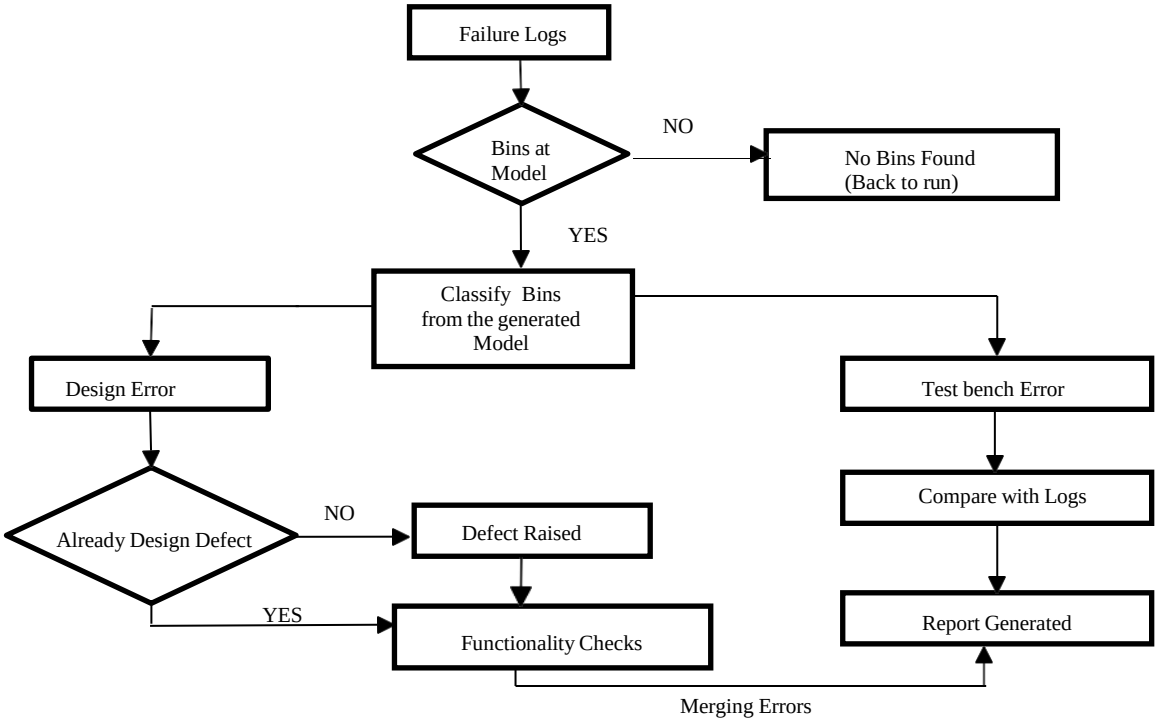


Fig. 3.4: Flowchart

3.4 Verification Methodology

Due to the intricate nature of modern chip architectures and the evolving demands of industry, there is a need of advancements in the verification process. Various approaches are employed to address these challenges, including UVM based and assertion based methodologies.

3.4.1 SV- UVM Based Methodology

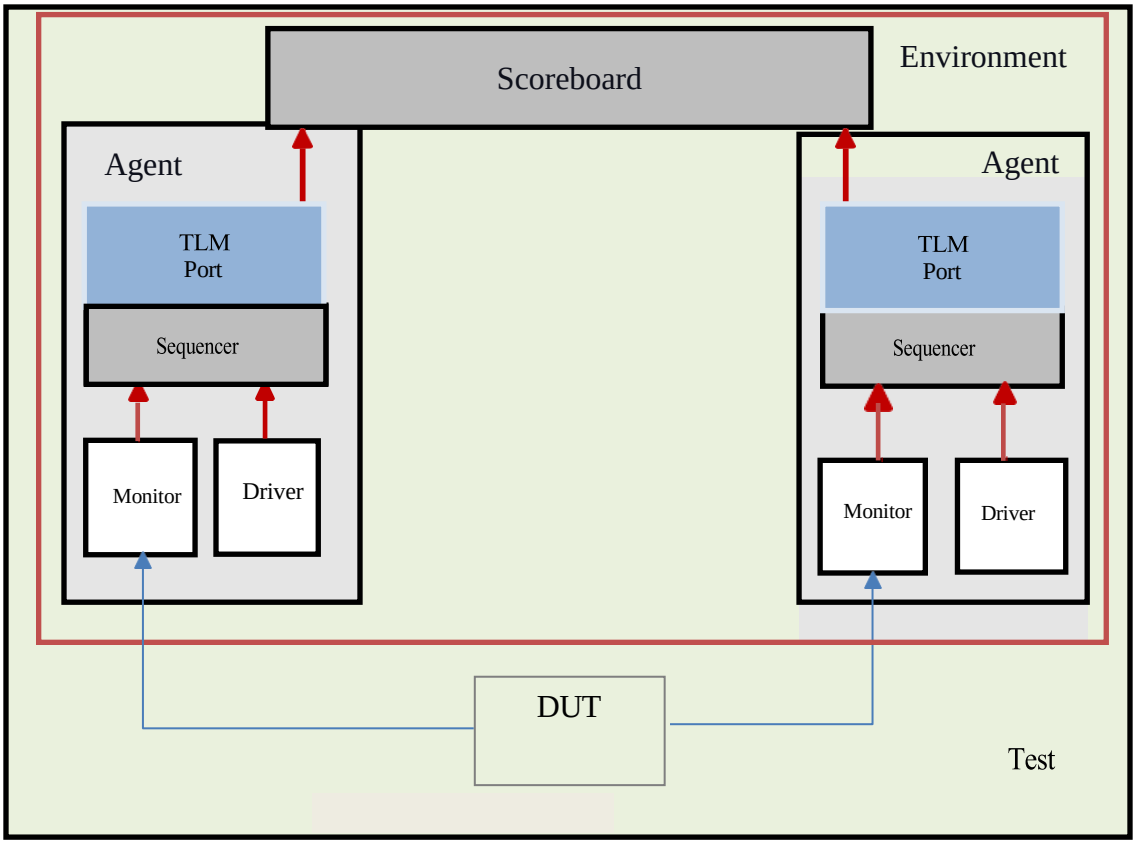


Fig. 3.5: UVM Based Methodology

SV integrated with the new verification methodologies using object and components oriented concepts, significantly increasing the efficiency of the verification process. It facilitates an environment taking classes and randomization constraints. Within this framework, DUT is linked to the agent and monitors with its sequencer. Other key components include drivers,

monitors, agents, and top-level modules are also there where each of these components and its objects are doing their specific roles as per the specifications.

UVM is generally used due to its reusable test bench environments which are created with the help of predefined classes. This is a basic standard of industry written with the help of SV to create an efficient verification environment. UVM uses a predefined class with the factory mechanism without altering any other file of the test bench, only the classes and objects are registered and gets whenever needed.

3.4.2 Assertion based Methodology

Assertions serve as critical tools in verifying whether a design adheres to specified conditions. When an assertion fails, it indicates a deviation from expected behavior, required further investigation into the underlying causes.

Two types of assertions statements are used, which are as follows:

i. **Immediate Assertions**: It is used to check different conditions at current time.
   Example: a_eq_v: assert (a ==v) else $error (“a not equal v”);

ii. **Concurrent Assertions:** It examines sequence of events over the multiple cycle that focus on evaluating the occurrence of specific sequences of events across multiple clock cycles.
   Example: p_eq_q: assert property ((@posedge clk) c |=> (p == q));

3.5 IP Interfaces

In high speed data compression, there are several interfaces used for functionalities and the clock reset unit for the blocks used at 400 MHz target frequency.

3.5.1 APB Interface

This technology relies on the AMBA which is standard for managing register field bits of internal register. It offers an efficient way to integrate protocols that minimizes power consumption. This interface uses defined protocols to connect low bandwidth peripheral with assigned masters. Bigger devices are connected with the interfaces like AHB and the small

operating peripherals are connected with the interfaces APB.

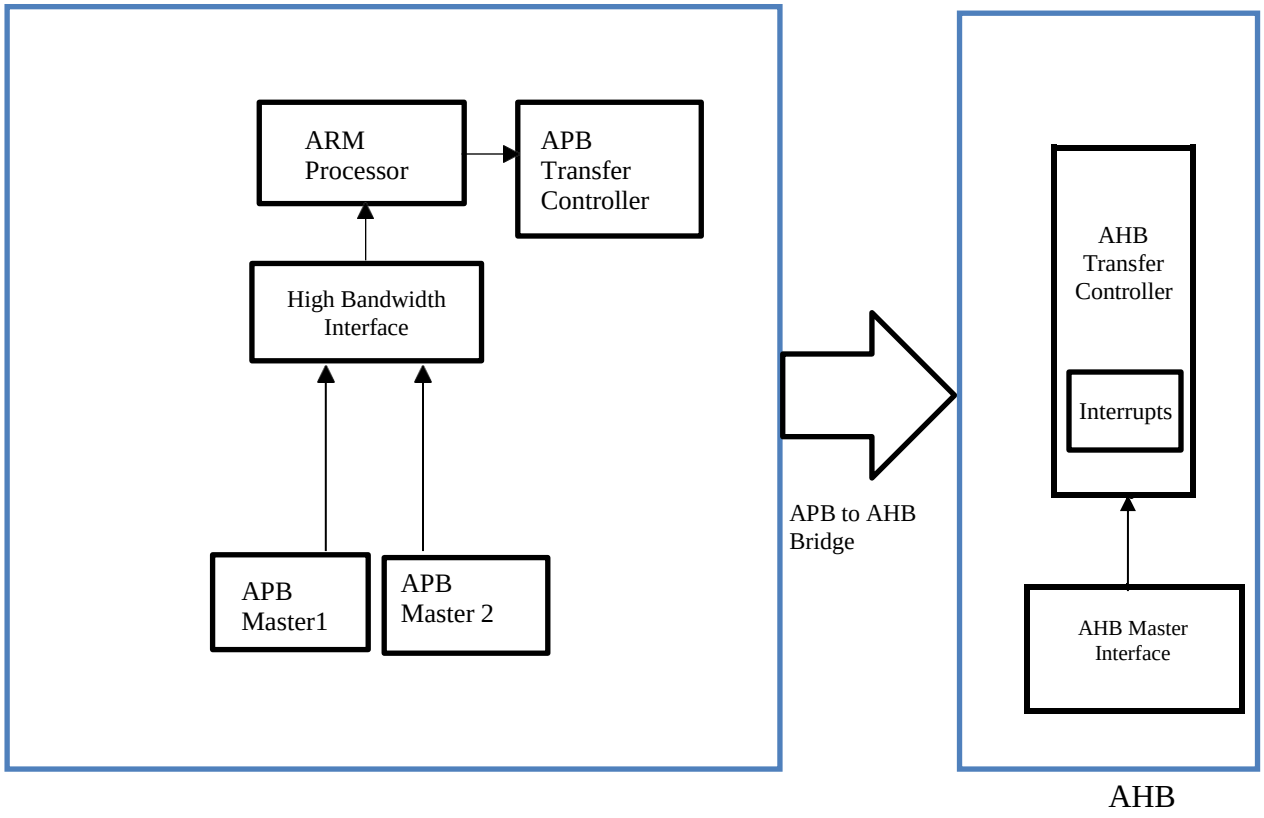


Fig. 3.6: APB to AHB bridge flow

The operating states of APB protocol are as follows.

i. IDLE: This is the initial state of APB protocol. In this state the reset condition is set up. This is the default state for APB where no transition occurs.

ii.SETUP: When the setup phase starts, the transition occurs when a transfer is initiated and the PSELx signal is asserted. This phase lasts for a single clock cycle, after which the state transitions to the next phase during setup at the first rising edge of the clock.

iii.ACCESS: During the access phase, the PENABLE signal is asserted, indicating the readiness for data transfer. It is crucial that signals such as data, and address remain stable as the system transitions from the setup phase. If PREADY remains low, the bus stays in the access state and if the PREADY signal asserted then the bus goes to the idle state if no transfer is required. If interrupt occurs then bus goes to stop the execution process.

Handshaking mechanisms utilize the ATVALID and ATREADY signals to fetch the data by asserting the ATVALID signal. If the slave is ready to accept data, it responds with the ALREADY signal else the ATREADY signal remains low. Hence this communication protocol involves the master asserting the ATVALID signal to indicate data availability, and the slave responding with the ALREADY signal when it receives the incoming data.

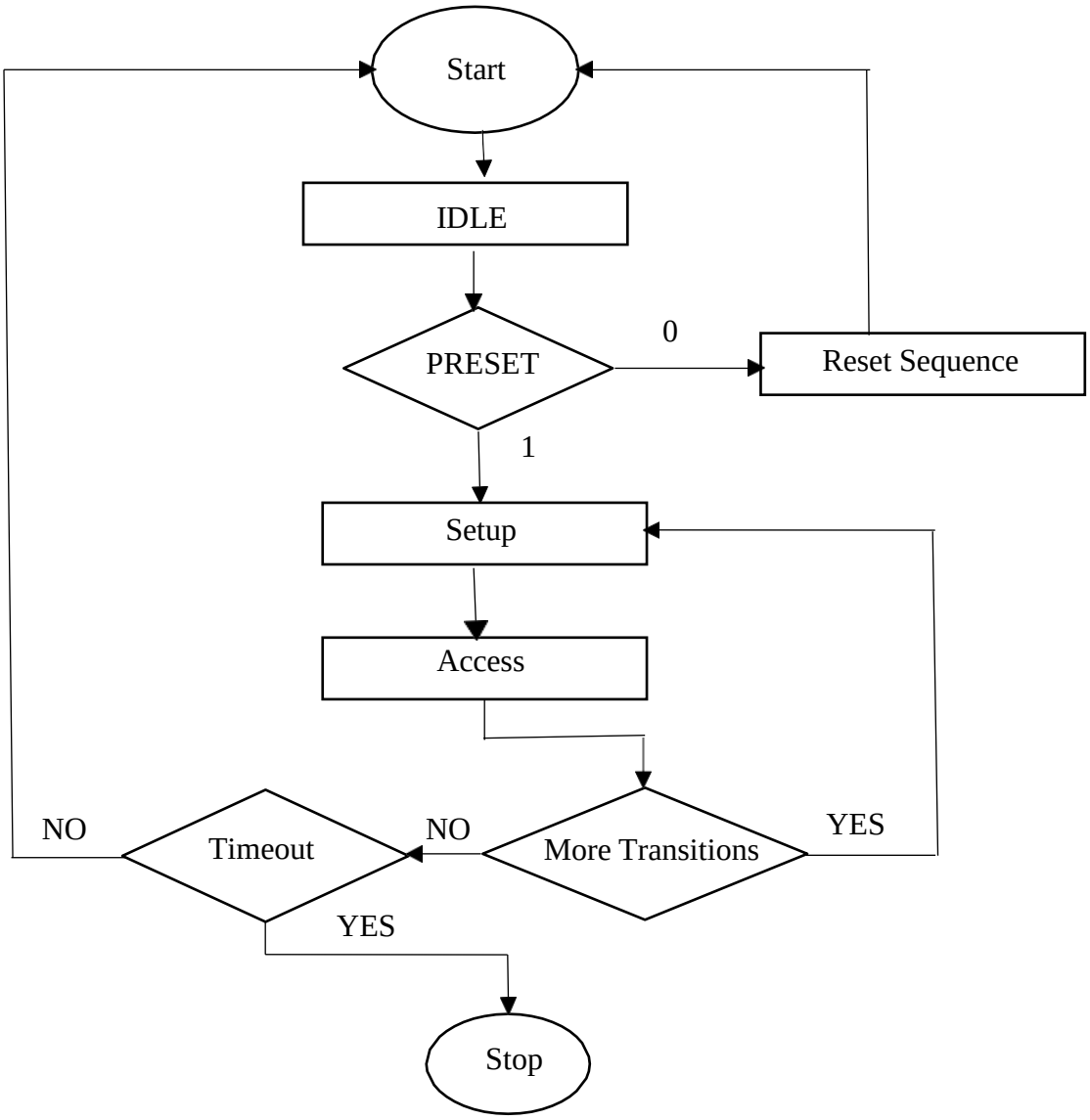


Fig. 3.7: APB State Diagram

Now if the signal ATREADY is low and ATVALID is high in a cycle, then the signal ATDATA has some value and in case ATVALID is low in a cycle then the condition of ATREADY doesn't provide any effects. There are many times when it is imperative to remove remaining data from the buffer, this process is called flushing.

The number of pipeline stages is generally fixed in the master. Whenever a trace is generated, it is stored in a data locations and data is coming out from the FIFO.

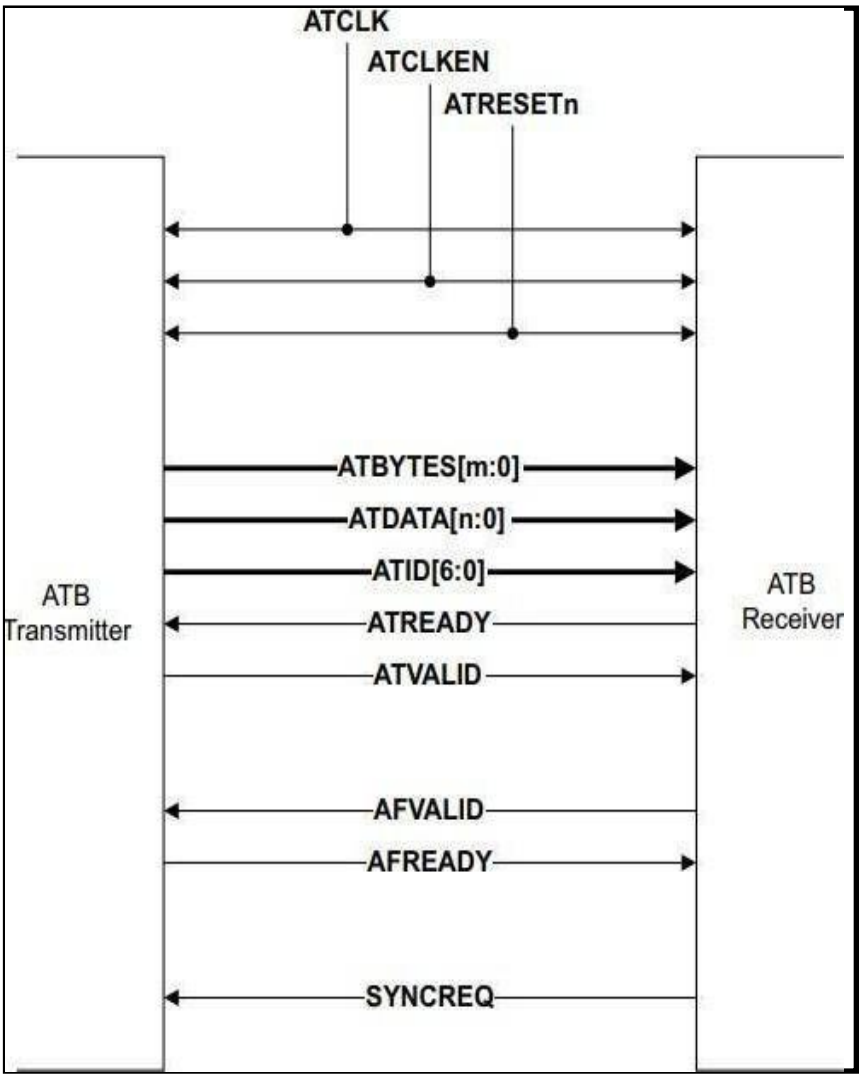


Fig. 3.8: Different ATB Signals

As shown in the waveform in figure 3.9, ATDATA is 32 bit and its byte value is 2 BITS and the bit values are taken as per the requirement of the protocol. The below waveform gives the analysis of single state.

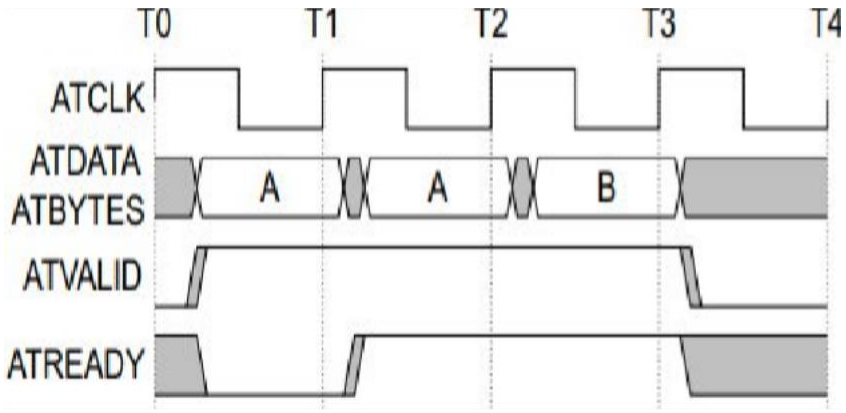


Fig 3.9 Waveform analysis for single state

At T1 clock cycle ATREADY is not accessed, at T2 clock cycle value B is asserted and ATREADY is de-asserted, at T3, B value is asserted and at T4 clock cycle both A and B are de-asserted. Data stream is associated with a unique ID that distinguishes it among different sources and finds components with its objects for similar requirements. Signals such as ATBYTES and ATDATA share a relationship defined by the values in ATDATA[m] and ATBYTES[n].

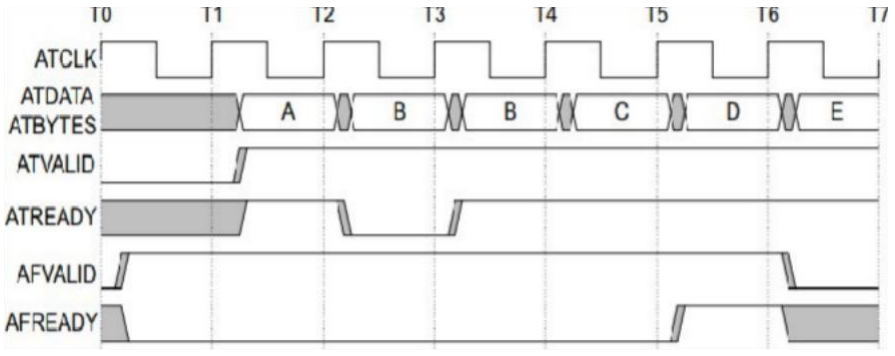


Fig. 3.10: Waveform Analysis for multiple States

Based on the waveform analysis for multiple states as shown in figure 3.10, it has been observed that AFREADY becomes high when all the trace data generated while ATREADY and AFVALID is asserted and data transferred. The assertion of AFREADY imply that the data processing running in background. ATVALID asserts high during the complete cycle after the generation of data in T2 cycle. The trace data generated after AFVALID is asserted in the FIFO and de-asserted when AFREADY becomes active.

Trace data which is generated after the assertion of AFVALID signal is stored and then AFREADY is asserted. At TI-T2 clock cycle, flush is requested and values are held in the FIFO. Now at clock T2 flushing starts and at T3-T5 clock stalls but data transfer is still possible and at T6 clock flushes out and AFVALID signal becomes low which makes asserting high of AFREADY signal just before one cycle.

Major advantages of using RAL are:

i. Reusability: They are able to be reused in other environments and also enable users to write sequences to access field registers and regions of memory.

ii. Uniformity: Across the semiconductor industry the set of pre-defined rules on register access are common.

The process of generating the RAL model follows a certain process which is command based and automated using standard tools. Initially, the design team provides a XML file containing register descriptions of a register file which is intended for use the specifications in the RTL design. This XML file is then placed using fetching commands into the RAL model generation tool and it generates the RAL blocks model having register fields. These essential register fields tasks are encoded into RAL sequences, facilitating standard verification across all the RAL models.

This approach narrow down the process, allowing for register read/write test cases to be executed and analyzed. It is easier to access registers with the help of RAL model as it classifies the bits into multiple register fields which creates the bit access of particular register

values. The RAL model sequences automate the register field verification tasks ensuring efficient validation of registers functionality.

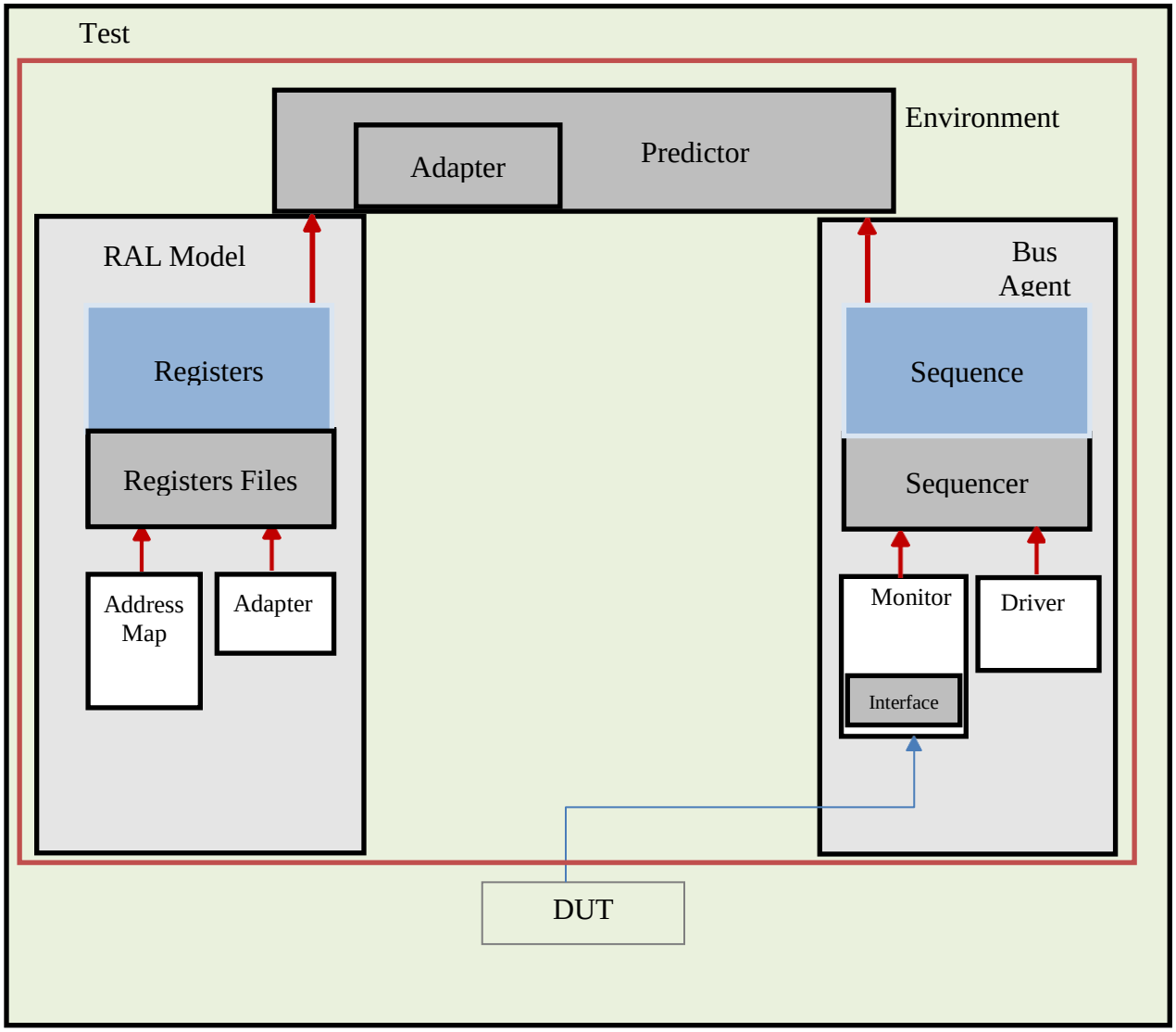


Fig. 3.11: RAL Model

3.6 UVM Component

A UVM is consists of a component that are connected as an elements and used by test bench for verifying IPs. These components are layered to handle different aspects of the verification process, such as signal displaying. The main UVM components includes “uvm_test,” which

configures the verification environment, “uvm_env” which defines the overall test bench architecture, “uvm_agent”, responsible for doing the protocol tasks, and “uvm_driver” and “uvm_monitor,” which generate test stimuli and observe the outputs of the DUT.

Figures marked with an asterisk (*) contain restricted data related to module names and are subject to company confidentiality.

3.6.1 UVM Sequence and Sequence item

The sequences are implemented using the UVM sequence class, to stimulate the DUT, sequence items are designed with data fields that model the test scenarios. Custom items are derived from the UVM sequence item class which are useful for carrying functions contains specific items as per the specifications which streamline debugging processes. It generated the data fields as per the verification plans developed by the verification teams. These data fields are typically randomized with some useful constraints to make test conditions, and to the randomized nature of the test bench environment.

Randomization is done to make the bit fields like address and data more variable with the help of constraints defined for specific functions as inline, soft, solve-before, bidirectional as per the requirements of the fields.

```
// Define a uvm_sequence_item
class my_seq_item extends uvm_sequence_item;
  rand bit [31:0] data;
  rand bit [9:0] addr;

  function new (string name = "");
    super.new(name);
  endfunction
  `uvm_object_utils_begin( my_seq_item )
    `uvm_field_int( data, UVM_ALL_ON )
    `uvm_field_int( addr, UVM_ALL_ON )
  `uvm_object_utils_end

endclass : my_seq_item
```

Fig. 3.12: Sequence

3.6.2 UVM Sequencer

The sequencer is created by extending the UVM sequencer class with the item as a parameter. It generates and returns data items in response to the driver's commands. Using randomization, a sequencer follows the present condition of the DUT and generates specific conditions sequences which contain more valuable data items.

```
class apbSequencer#(type PARAM=int) extends uvm_sequencer
  typedef apbSequencer#(PARAM) apbSequencer_t;
  `uvm_component_param_utils(apbSequencer_t)

  function new(string name, uvm_component parent=null);
    super.new(name,parent);
  endfunction


endclass : apbSequencer
```

Fig. 3.13: Sequencer

3.6.3 UVM Driver

The driver in semiconductor testing acts as an intermediary between the test environment and the DUT. The driver which operates in two modes is specifically designed to create customizable drivers. This parameterized class incorporates functions for sequence items to define the types of data sent to the DUT during testing.

```
class apbDriver #(type PARAM=int) extends uvm_driver #(a

  typedef apbDriver#(PARAM)       apbDriver_t;
  typedef apbTransaction #(PARAM) apbTransaction_t;
  typedef virtual interface apb_if#(PARAM) apb_if_t;
  `uvm_component_param_utils(apbDriver_t)

  bit reset_done;
  bit packet_received;
  apb_if_t apb_if_h;

  function new(string name, uvm_component parent=null);
    super.new(name,parent);
  endfunction
```

Fig. 3.14: Driver

3.6.4 UVM Monitor

This component operates independently to monitor all interactions between the DUT and the test bench. It forwards signals to other modules like scoreboard, if communication deviates from the protocols, the monitor flags errors else it translates signal level data into sequence item format from transmitting data to the scoreboard.

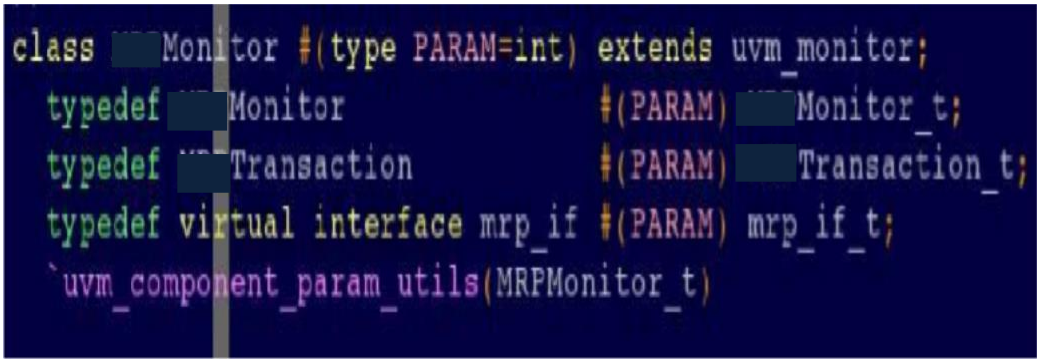


Fig. 3.15*: Monitor

3.6.5 UVM Agent

The UVM agent, is used by user-defined protocols in the test environment. It consolidates essential elements like drivers, sequencers, and monitors. Multiple agents exist within a single UVC, catering to various protocols in SoC designs. The agent connects through stages to interconnect components which are distinct from the running phase. Agents operate actively or passively based on their requirements.

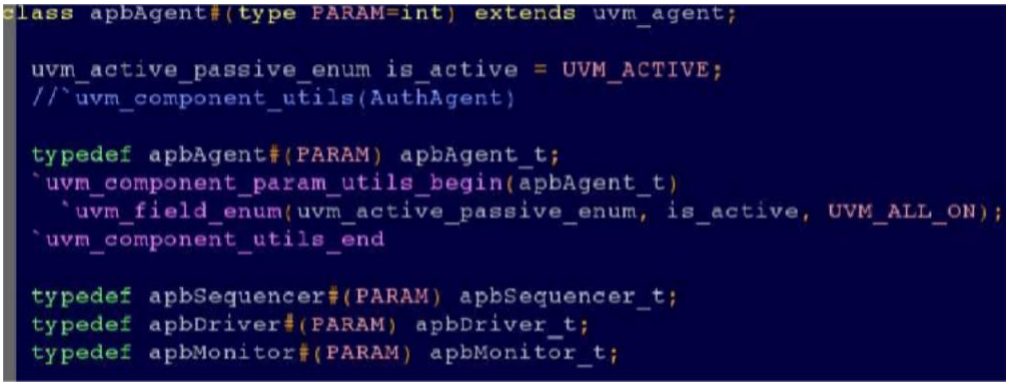


Fig. 3.16: Agent

3.6.6 UVM Scoreboard

A scoreboard in verification ensures the correct operation of the DUT by monitoring the flow of data into and out of it. It compares the inputs entering the test environment with the outputs produced by the DUT. The scoreboard collects data from multiple sources, processes it independently, and compares data with the expected results to validate the DUT.

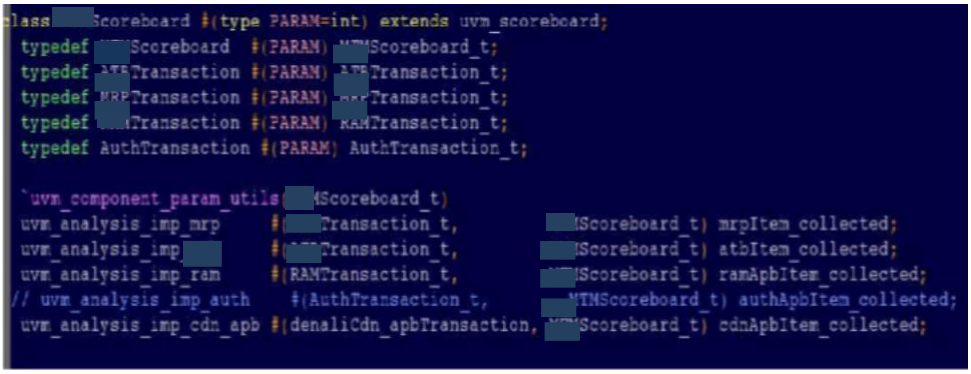


Fig. 3.17*: Scoreboard

3.6.7 UVM Environment

At the top level, this component acts as a central hub for managing multiple agents. Built upon the UVM component base class, it coordinates diverse verification tasks. It includes functionality such as scoreboards for validating data flows and comparing outcomes to reference models. Furthermore, the component integrates previously verified block-level environments into larger subsystems or SoC architectures.

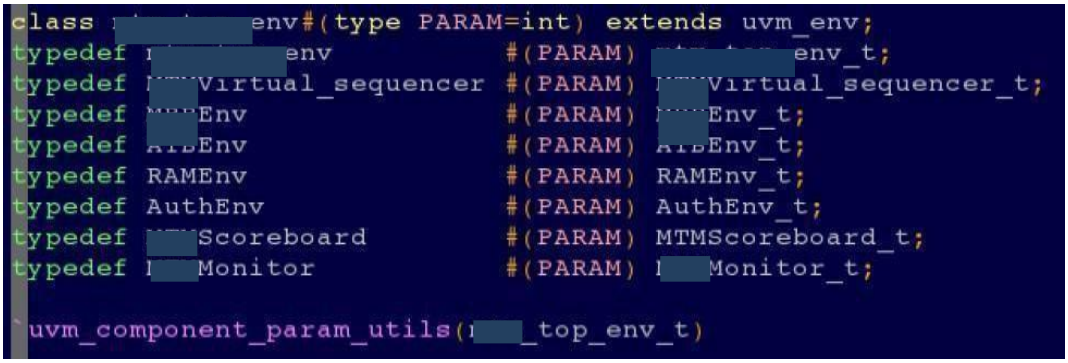


Fig. 3.18*: Environment

3.6.8 UVM Test

Test generates the test bench scenarios by carrying out the sequencer to define sequences generation for a particular run allowing the data items to transmit to the DUT based on their requirements. Test is written as per the basic specifications given by the design team and verification plans made by the verification teams. They are written to check the specific scenarios of particular IP/SoC where SoC handshaking process of C and SV is done to make a verification of functionalization of the test.

3.6.9 UVM Top

It is the block where DUT and test bench instances are built and connects the DUT to the test bench using the virtual interface as a handshaking process. It contains the SV files for various components, UVM packages, and sometimes certain assertion definitions.

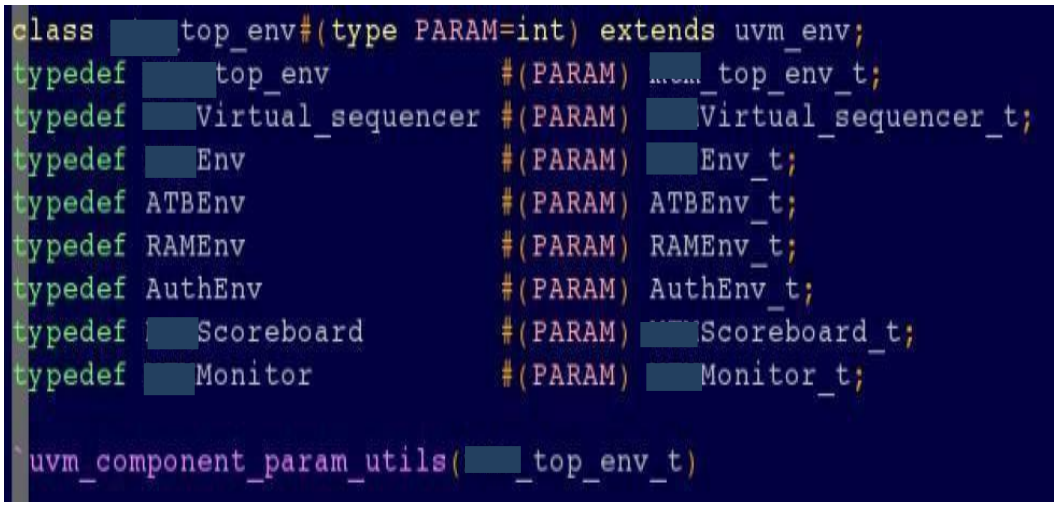


Fig. 3.19*: Top Module

3.7 Safety IP

The Safety IP core module is a unit which controls access to target regions based on configure access permission. It interacts with the control unit, a NOC hardware unit that is instantiated within the interconnects of NOC. They can be in a range of a single bridge component to a complex architecture for specified protocol bridge which consists of multiple masters and slaves used to transfer the signal from interface to the specified modules of the blocks.

The protection mechanism is based on the use of software defined protection regions that enable fine-grained access control at the target based on the following attributes of each request transaction. This is divided into checks based on below attributes: as memory type, master type, Initiator address, Initiator master ID access type (Read/Write/Exec) access qualifier. It supports an APB4 interface for the hardware configurability.

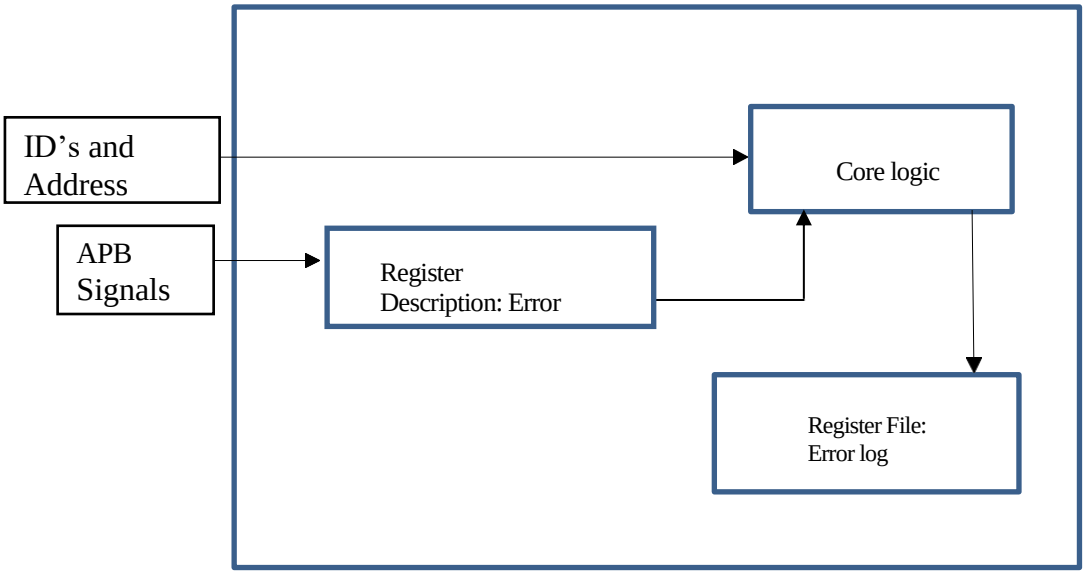


Fig. 3.20: Safety IP top-level block diagram

3.7.1 Key features

Safety IP has some features for easy access to the regions in the architecture. It has a mechanism to control access to multiple partitions of the IP. The key features of safety IP are as follows:

i. Possibility to enable/disable the Safety IP checks.

ii. Read/Write/execute access permissions of each Initiator for protection region.

iii. Support communication between virtual machines using the shared memory.

iv. Shareable attribute per region for access checks.

v.All regions are parametrized and can go up to the maximum of 32 bits.

3.7.2 Configuring the Safety IP

The process begins by define the specifying bus protocols, widths and establishing master-slave connectivity matrices. Designers set up the connectivity topology in the top implementation view. The tool graphical interface allows for selecting specifications such as buffer depths and register settings with their configuration options.

Designing an IP involves navigating complexity to ensure a bug-free outcome, making defect raised which is critical in semiconductor design. Verification states the design's functional accuracy against specified requirements and this functional verification begins by validating the design's behavior. RTL designers collaborate with verification teams using test benches and coverage metrics to validate the code. Key metrics include expression and other coverage, aimed to achieve coverage range through code-based assertions verification.

3.7.3 Environment Setup for IP Verification

Once the register test cases are planned by the architecture team, the verification team first set up the environment for IP verification. In this situation, a dedicated data transmission is crucial at the SoC level to effectively handle data reception to ensure its comprehensive functionality. Verification IPs are reusable components that are integral to the test bench framework and sourced either through in-house or third-party developed, the current IP verification test bench utilizes wrapper APB VIPs, which are essential components in our verification methodologies. Verification of RTL requires building an elaborate environment set-up for running tests, identifying regressions and arranging results from test-runs. Creating this environment for every RTL is a manual task for each team member and it may take weeks of effort. The abstraction of the register model closely resembles the structure for RAL register definitions. This abstraction serves as a pivotal reference point for verification teams, and design teams. The critical aspect is the accuracy of this model to effectively validate the design.

This model, subsequently used for accessing DUT registers, unifies efforts across different teams and ensures thorough verification of the IP designer. Repeating the above steps is a mundane, repetitive task, which is carried out manually and is prone to human errors. The Environment

setup is to generate a test bench environment for a specific protocol. It takes input from a configuration file and generates a test bench directory with all the basic files. It also connects the DUT to the test bench. It fetches signals and other information from XML and uses that information to automate setup. It also generates standard register verification test cases.

Chapter 4

Results and Discussion

The outcome enhances the efficiency of regression testing by maximizing its utilization, directing manual effort towards genuine debugging tasks rather than other tasks. This approach also reduces the time for the debugging process in regression testing for chip projects. It leads to significant increase in productivity for each debugged test, thereby reducing the number of overall tests to be developed.

```
# Building a decoder
def decoder(conv4):
    conv5 = Conv2D(128, (3,3), activation = 'relu', padding = 'same') (conv4) # 7 * 7 * 128
    conv5 = BatchNormalization()(conv5)
    conv5 = Conv2D(128, (3,3), activation = 'relu', padding = 'same')(conv5)
    conv5 = BatchNormalization()(conv5)
    conv6 = Conv2D(64, (3,3), activation= 'relu', padding ='same') (conv5)# 7 * 7 * 64
    conv6 = BatchNormalization()(conv6)
    conv6 = Conv2D(64, (3,3), activation= 'relu', padding = 'same')(conv6)
    conv6 = BatchNormalization()(conv6)
    up1 = UpSampling2D((2,2)) (conv6)  # 14 * 14 * 64
    conv7 = Conv2D(32, (3,3), activation = 'relu', padding = 'same') (up1)# 14 * 14 * 32
    conv7 = BatchNormalization()(conv7)
    conv7 = Conv2D(32, (3,3), activation ='relu', padding = 'same')(conv7)
    conv7 = BatchNormalization()(conv7)
    up2 = UpSampling2D((2,2))(conv7) # 28 * 28 * 32
    decoded = Conv2D(1,(3, 3),  activation = 'sigmoid', padding ='same') (up2) # 28 * 28 * 1
    return decoded

# Compiling the model using RMSProp optimizer (helps in reducing overfitting by
# changing learning rate while the model is training) and specifying the loss as mean squared loss
autoencoder = Model(input_image, decoder(encoder(input_image)))
autoencoder.compile(loss='mean_squared_error', optimizer = RMSprop())

# Let us see the autoencoder summary
autoencoder.summary()
```

Fig: 4.1 Convolution Layer Generation

Fig. 4.1 shows the generation of the Convolution layer. It comprises of a set of filters, where each filter's parameters used neural network and layers are adjusted to optimize the performance. These filters convolve across the dimensions of the input layers at its each spatial position. This operation generates activation maps that highlight specific features across the input matrix.

Sigmoid function can be mathematically represented as $\sigma(k) = 1/((1+e)^{-}k)$ (4.1)

where $\sigma(k)$ is sigmoid function k is the number of input values

ReLU function can be mathematically represented as $f(r)=\max(0,r)$ (4.2)

where f(r) is ReLU function r is the number of input values

This layer generation process works by initially extracting the features from the developed matrix followed by building a decoder to decode the extracted features into multiple layers. Initially, the input convolutional layer along with different activation functions like Relu and sigmoid function are normalized. The use of the sigmoid function is to convert all the negative values to positive values using the equation 4.1 so that mapping can be done for the dataset properly and the Relu activation function is used to get the linearity of the function using the equation 4.2. Hence, the initial layer as shown in figure 4.1, “conv5” and “conv6” uses the activation functionReLU and the last layer “conv7” uses the activation function sigmoid. In the last layer, decoded values are generated and structure is mapped for required dimensions of length and width (3, 3)respectively. The “conv7” layer is required to send flat data to a neural connected layer. In this layer, data is transformed into the variable classes to get a fully connected network. For the index value of 1 and with size 3*3, a layer is generated for the compilation.

Categorized autocross entropy uses the calculated mean square error by taking the model optimizer value to reduce the overfitting. Sampling is done in the training data for our model by considering the batch size of 128, it means that it takes 128 values at once and this process is repeated until all the fetched parameters are not satisfied.

Fig. 4.2 shows model training by building an encoder and a decoder using convolutional layers. It is a function that constructs the decoder part of a convolutional encoder. The decoder is basically used for construction of input matrix which is responsible for generation of results using input data. In the training model, the layer named as “input_1” has been created with the length, width and layer index that is (28, 28, 1) respectively.

First, second and third parameter values (128, 128, 256) are considered as the value of the height, width and number of output channels. Output filters of 32 bit are taken for a (3, 3) feature extraction. Padding used is "same", which processes only the even padding for the input shape of (128, 3, 3). The max-pooling layer is used for the pool size of (7, 7) that defines the pooling window. Training data is sampled for 12 epochs initially it means 12 times it processed the

model training. In the “input_1” layer when 32 filters of size (3 * 3) are applied then the output shape becomes (28, 28,1).

Convolutional layers are used with the filters of size 3x3, ReLU activation, and batch normalization methods. These layers are designed to perform features extraction. A convolutional layer with a single filter is used to reconstruct the original input data.

New parameters value is calculated with the following equations:

new_Param = old_Param - (learning_rate * gradient of Param#) (4.3)

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 28, 28, 1)	0
conv2d_1 (Conv2D)	(None, 28, 28, 32)	320
batch_normalization_1 (Batch	(None, 28, 28, 32)	128
conv2d_2 (Conv2D)	(None, 28, 28, 32)	9248
batch_normalization_2 (Batch	(None, 28, 28, 32)	128
max_pooling2d_1 (MaxPooling2	(None, 14, 14, 32)	0
conv2d_3 (Conv2D)	(None, 14, 14, 64)	18496
batch_normalization_3 (Batch	(None, 14, 14, 64)	256
conv2d_4 (Conv2D)	(None, 14, 14, 64)	36928
batch_normalization_4 (Batch	(None, 14, 14, 64)	256
max_pooling2d_2 (MaxPooling2	(None, 7, 7, 64)	0
conv2d_5 (Conv2D)	(None, 7, 7, 128)	73856
batch_normalization_5 (Batch	(None, 7, 7, 128)	512
conv2d_6 (Conv2D)	(None, 7, 7, 128)	147584
batch_normalization_6 (Batch	(None, 7, 7, 128)	512

Fig. 4.2 Model Training

The sigmoid activation is often suitable for training purpose with it functions to place normalization of inputs in the suitable format to convert the batches into defined matrix for accuracy calculation. It gives model accuracy of 91.9 percentwhich is normalized by the loss of 0.3 percent as shown in Fig. 4.3.

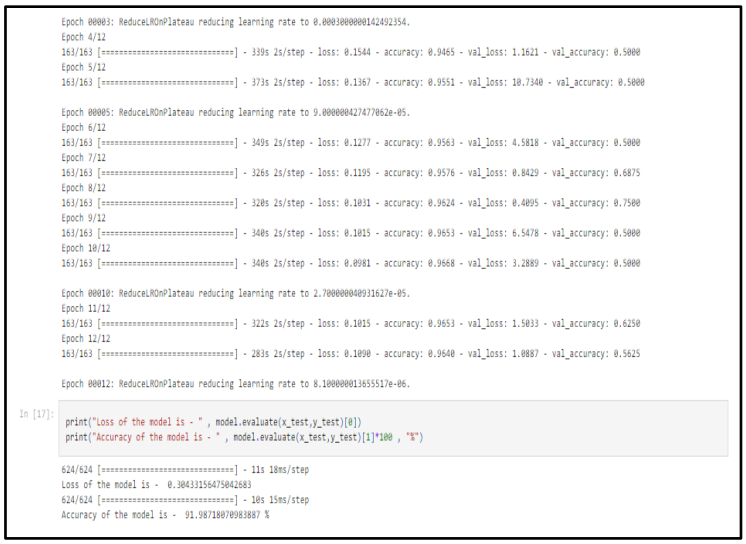


Fig. 4.3 Model Accuracy Result

The model is trained using the K-means algorithm over several epochs. To ensure accuracy, the dataset is divided into multiple sets, with 80% assigned to train purpose and 20% to testing purpose.

The test set acts as a subset to evaluate the model's performance after training. A predefined threshold of 75% accuracy is set to guide the training process, ensuring the model continues training until this desired level of accuracy is achieved. Confusion matrix provides a detailed breakdown of the model's accuracy by displaying the counts of true negatives and also false positives and negatives.

Bins	Testcases	Context Field	Severity	Actual debug	% Accuracy
1	26	RCU_fault	High	19	75
2	21	ACU_fault	High	20	96
3	8	CCR_mismatch	High	13	62
4	12	Timeout Errors	Low	12	100

Table. 4.1 Bins Accuracy Result

Table. 4.1 compares the multiple bins test cases with the actual test cases failures that are failed due to the same root cause contexts. It shows the accuracy of failing test cases with the actual test cases debugged by the verification team. It gives a 100% result for the low severity errors such as timeout errors. For the high and medium severity errors, the accuracy ranges from 60 to 90 percent. For faults categorized into the bins as per the given context field the number of test cases lies in the range of 10 to 25 test cases.

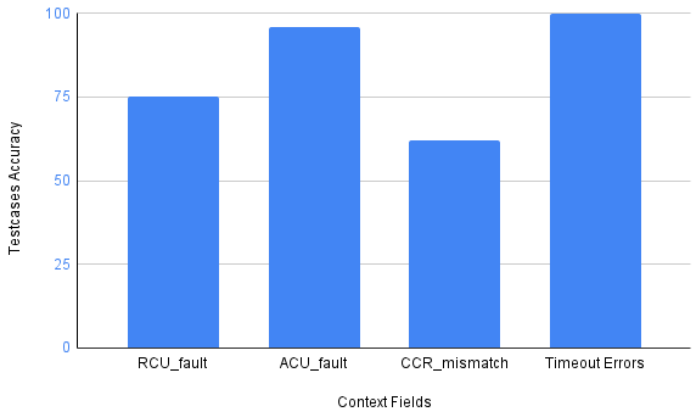


Fig. 4.4 Bins Accuracy Comparison

Fig. 4.4 represents the graphical form of test cases clustered in multiple bins as given in the above table. It compares test cases with the actual test cases that failed due to the same root cause contexts. The accuracy of bins is compared, revealing that bin 1 has the highest number of failing test cases, exceeding 25, while bin 3 exhibits the lowest with fewer than 10 failing cases.

This analysis clearly indicates that bins associated with higher severity contexts have a lower likelihood of achieving accuracy, whereas bins linked to lower severity contexts are more likely to meet accuracy standards.

Fig. 4.5 depicts the accuracy achieved in the testing phases. Before testing a model, raw data undergoes initial analysis and preprocessing to simplify the model-training process and handle any unclustered data effectively. The accuracy levels observed during the testing phases

significantly influence the final score. The parameter Win specifies the slices involved in the operation and epochs are measured for accurate values with threshold value defined in the model. It calculate the accuracy within the function of referenced model but error propagations can occur during generation of accuracy.

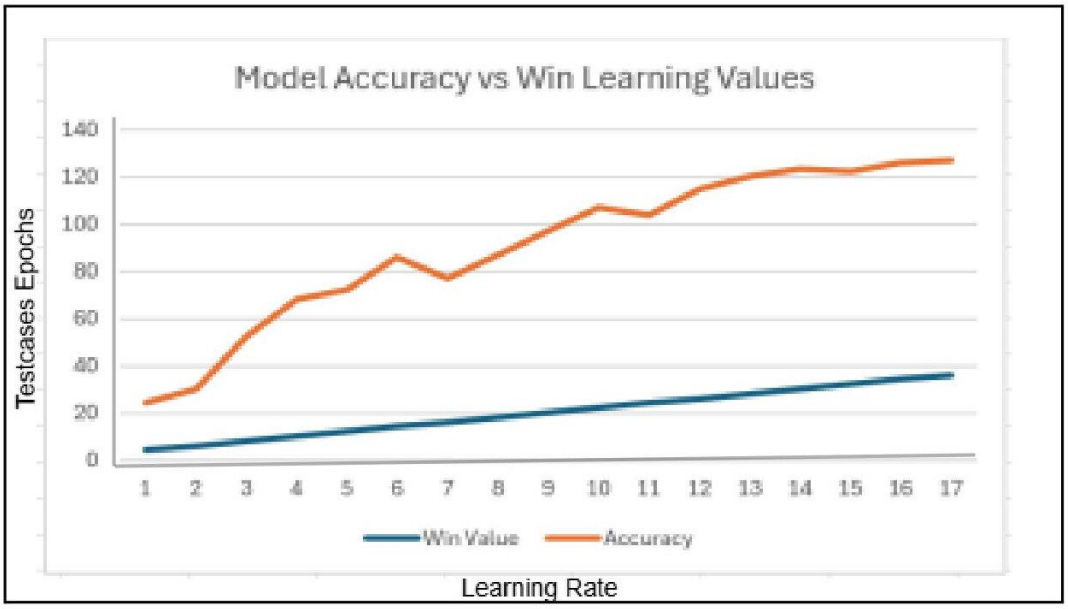
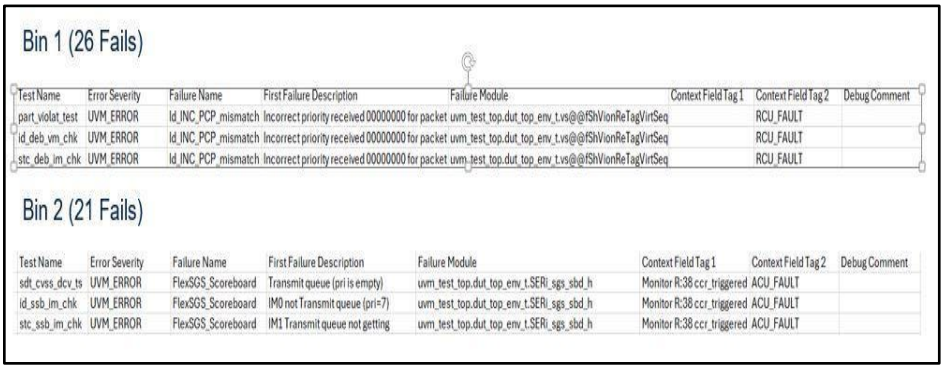


Fig. 4.5 Model Accuracy vs Wn Values

Fig. 4.6 shows the output file having multiple bins with its test failures. It categorized the testcase into multiple bins using failure names, error severity, failure description, and modules. It also includes context fields or specific tags found given by the user.



Bin 1 (26 Fails)

Test Name	Error Severity	Failure Name	First Failure Description	Failure Module	Context Field Tag 1	Context Field Tag 2	Debug Comment
part_violat_test	UVM_ERROR	Id_INC_PCP_mismatch	Incorrect priority received 00000000 for packet	uvm_test_top.dut_top_env_t.vs@@fShVionReTagVirtSeq		RCU_FAULT	
id_deb_vm_chk	UVM_ERROR	Id_INC_PCP_mismatch	Incorrect priority received 00000000 for packet	uvm_test_top.dut_top_env_t.vs@@fShVionReTagVirtSeq		RCU_FAULT	
stc_deb_im_chk	UVM_ERROR	Id_INC_PCP_mismatch	Incorrect priority received 00000000 for packet	uvm_test_top.dut_top_env_t.vs@@fShVionReTagVirtSeq		RCU_FAULT	

Bin 2 (21 Fails)

Test Name	Error Severity	Failure Name	First Failure Description	Failure Module	Context Field Tag 1	Context Field Tag 2	Debug Comment
sdt_cvss_dcv_ts	UVM_ERROR	FlexSGS_Scoreboard	Transmit queue (pri is empty)	uvm_test_top.dut_top_env_t.SERi_sgs_sbd_h	Monitor R:38 ccr_triggered	ACU_FAULT	
id_ssb_im_chk	UVM_ERROR	FlexSGS_Scoreboard	IM0 not Transmit queue (pri=7)	uvm_test_top.dut_top_env_t.SERi_sgs_sbd_h	Monitor R:38 ccr_triggered	ACU_FAULT	
stc_ssb_im_chk	UVM_ERROR	FlexSGS_Scoreboard	IM1 Transmit queue not getting	uvm_test_top.dut_top_env_t.SERi_sgs_sbd_h	Monitor R:38 ccr_triggered	ACU_FAULT	

Fig. 4.6 Bins Failures Output

The output is generated in a CSV file format to analyze this output for debugging purposes. It saves significant time and effort for every debugged failing test while significantly reducing the number of such tests.

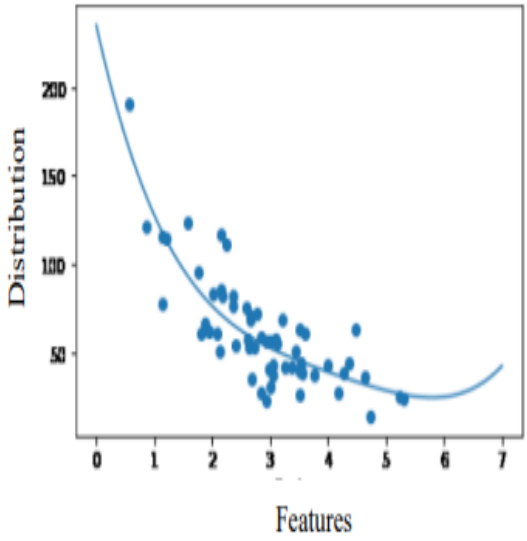


Fig. 4.7 Distribution of Samples

Fig. 4.7 shows the distribution of samples concerning failure names, error severity, failure description, and modules. The majority of samples fall within the range of 2 to 4 features, suggesting that samples with 2 to 4 common features are distributed in a classified manner. Therefore, it can be conclude that samples containing more features are effectively clustered. It also indicates that as the number of features increases, the distribution becomes less random, and clusters move closer together. This demonstrates a decrease in the gap between testing and training samples, suggesting improved alignment and similarity between the datasets used for testing and training purposes.

4.1Comparison of Results with existing Debugging Systems

This comparison evaluates the performance of existing debugging techniques, such as those using ML and bug analysis with proposed debugging systems. Parameters are focused on accuracy, bug type, design complexity and description level to assess these advancements offer significant improvements over existing methods.

Authors / Parameters	Eman El Mandouh et al. [18]	Mustafa Efendiog et al. [28]	Djordje Maksimovic et al. [14]	Ashima Kukkar et al. [8]	Proposed Approach
Accuracy	81%	96%	80-85%	85-99%	91.9%
Type of bugs	Similar Features Bugs	Core level Bugs	Register and Functional level Bugs	Duplicate Bugs only	Register, Functional, Core Level Bugs
Design complexity	Block Level	Single Core Level	IP level	SoC Level	SoC level
Description level	Logical and architecture	Logical	Logical and architecture	Logical	Logical and architecture

Table. 4.2 Comparison Results from existing approaches

Table 4.2 show the experimental results of proposed bug analysis approach and are compared with the existing systems based on similar data sets. These systems are proposed by Eman El Mandouh et al., Mustafa Efendioglu et al., Djordje Maksimovic et al. and Ashima Kukkar et al., [18], [28], [14], [8]. Eman El Mandouh et al. [18] optimize debugging for similar features bugs types for a block level at both logical and architecture levels with a accuracy of 81%. Mustafa Efendiog et al. [28] developed bug detection system using the combination of modeling and information retrieval process with 96% accuracy which is higher the our proposed system but it limits only for core level bugs. Djordje Maksimovic et al. [14] determines both functional and register bugs at IP level with a accuracy of 80-85% and Ashima Kukkar et al. [8] developed deeplearning based system which able to detect duplicate bugs only with a accuracy of 85-99% at SoC level.

Hence, it can be summarized as proposed approach improves the debugging process significantly rather than other existing approaches considering accuracy and types of bugs along with its complexity.

4.2 IP Verification Results

Waveform analysis is used in verifying the correct implementation of the APB protocol. By capturing and analyzing the waveforms of APB transactions, it ensures that the protocol operates correctly and efficiently. The methodology and results demonstrate the effectiveness of waveform analysis in identifying and addressing potential issues in APB implementations.

The signals encompassed in the testing include APB protocols and functional test cases, which verify both protocol adherence and specific IP signals such as reset, clock, read data, configure registers, write data, and specification selection states. Upon observing the waveform in figure 4.8, at address 0x100 and time-stamp as marked in red, it has noted that when “PWrite” and “PSel” are enabled, then “Enable” signal asserts high, indicating the proper writing of data onto the registers. This observation confirms that register read and write operations are functioning correctly according to the defined specifications.

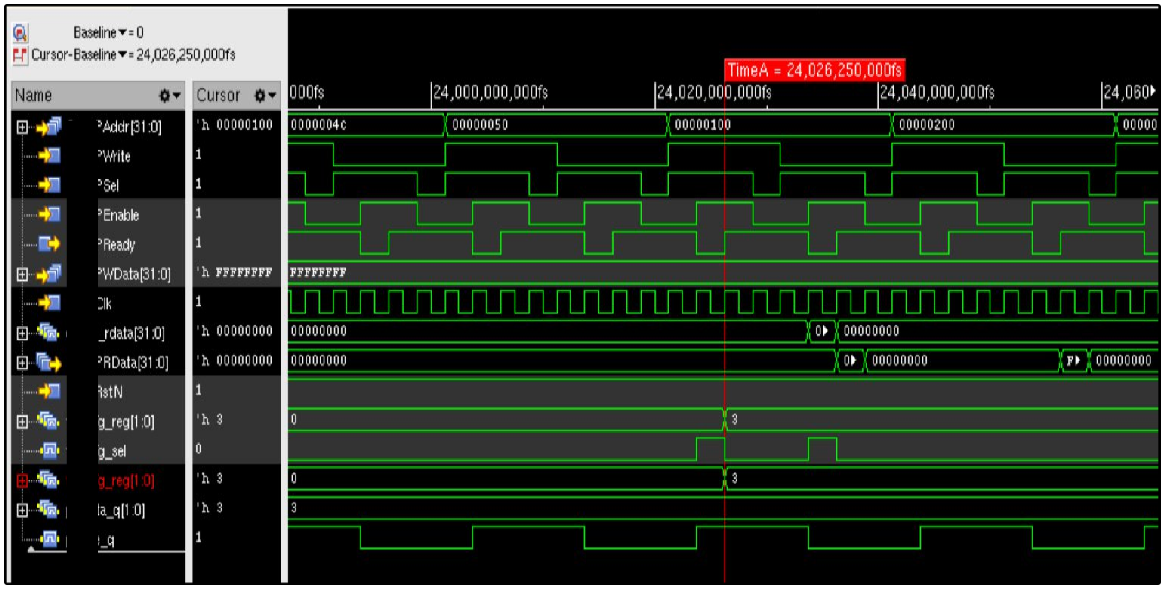


Fig. 4.8: Simulation result-1 Waveform Analysis

Signals numbered 11 to 15 from the top in the above waveform, “fg_reg [1:0]”, “fg_sel”, “wdata_q [1:0]”, and “write_q” serve the purpose of verifying the functionality of the IP. It is observed that when “fg_reg [1:0]” is in state 3 and “fg_sel” is asserted high, “write_q” also asserts high. Additionally, the red-marked signal in the above waveform, “f_cfg_reg [1:0]”, remains at state 3. These findings indicate that the safety configuration aligns with the expected functionality of the IP. This verification step confirms that the IP operates correctly according to its specified functional requirements under stated conditions.

Waveform analysis as shown in Fig. 4.9 confirmed that the APB transactions adhered to the protocol specifications. The signal transitions occurred as expected by the RTL. The “enable” and “ready” signals were correctly synchronized, ensuring proper data transfer. Additionally, the analysis highlighted potential areas for optimization, such as reducing the latency of certain transactions and improving the timing margins for critical signals.

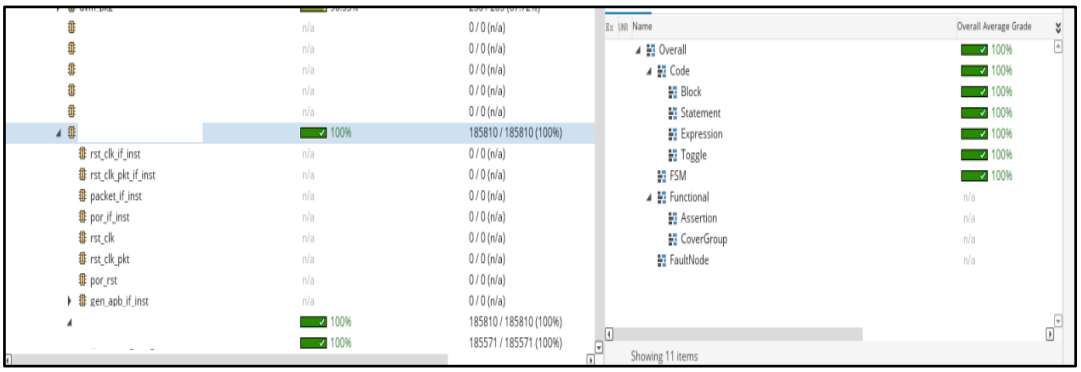


Fig. 4.9: Simulation result-2 Coverage Analysis

Coverage analysis is essential for ensuring the thorough verification of the APB protocol. By measuring toggle coverage, it ensures that all aspects of the protocol are exercised and validated. The results of this analysis demonstrate a high level of verification completeness, with identified gaps addressed through additional testing.

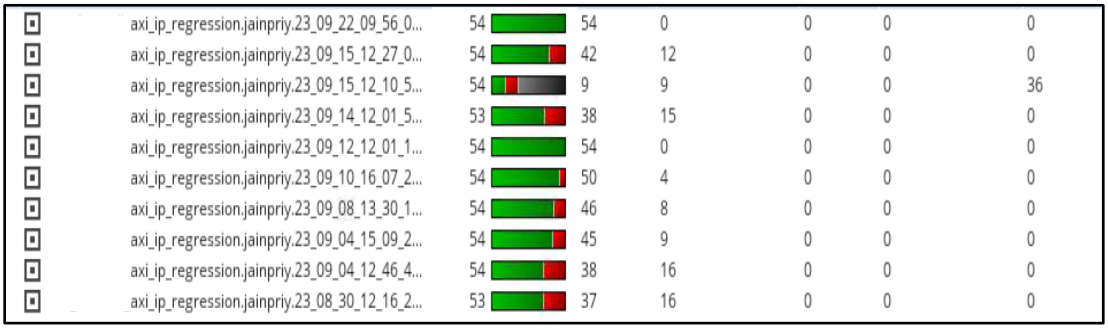
i. High Code Coverage: The goal is to achieve 100% statement coverage, branch coverage and path coverage. The insignificant uncovered areas are mostly related to error-handling paths not frequently exercised.

ii. Comprehensive Functional Coverage: Most functional scenarios were covered, including

normal and edge cases. A few edge cases related to rare timing scenarios required test cases.

Coverage analysis as shown in Fig 4.9 indicates that the top module, highlighted in blue, achieved a 100% status, confirming that all assertion properties validating the IP specifications are valid. A total of 185,810 properties were written to verify the RTL specifications, all of which passed successfully. Similarly, the core module had 185,571 properties written, and all these also passed validation. The combined verification results from both the core and top modules affirm that the IP has been comprehensively and functionally verified, meeting all specified criteria and ensuring its reliability for deployment.

The analysis highlighted some gaps in coverage, particularly in error conditions and certain edge cases. Additional test scenarios were developed to address these gaps to ensure comprehensive verification of the APB protocol.



axi_ip_regression.jainpriy.23_09_22_09_56_0...	54	54	0	0	0	0
axi_ip_regression.jainpriy.23_09_15_12_27_0...	54	42	12	0	0	0
axi_ip_regression.jainpriy.23_09_15_12_10_5...	54	9	9	0	0	36
axi_ip_regression.jainpriy.23_09_14_12_01_5...	53	38	15	0	0	0
axi_ip_regression.jainpriy.23_09_12_12_01_1...	54	54	0	0	0	0
axi_ip_regression.jainpriy.23_09_10_16_07_2...	54	50	4	0	0	0
axi_ip_regression.jainpriy.23_09_08_13_30_1...	54	46	8	0	0	0
axi_ip_regression.jainpriy.23_09_04_15_09_2...	54	45	9	0	0	0
axi_ip_regression.jainpriy.23_09_04_12_46_4...	54	38	16	0	0	0
axi_ip_regression.jainpriy.23_08_30_12_16_2...	53	37	16	0	0	0

Fig. 4.10: Simulation result-3 Regression Analysis

Regression analysis is essential for maintaining the integrity of the APB protocol implementation. After running multiple regressions tests it was found that failing test cases are getting passed. The results of this regression analysis demonstrate the effectiveness of the testing framework and provides roadmap for verification process.

The IP underwent functional testing with 54 test cases primarily focusing on ID violations, partition mismatches, register verifications, and secure gateways. Initially, only 37 test cases passed, contingent upon the "all state access" parameter being set. However, with the introduction of new regression tests, the number of failing test cases steadily decreased while

the count of passing test cases increased. Ultimately, after all regressions were completed, all 54 test cases passed, indicating comprehensive verification of the IP across all contexts. This final state confirms that the IP has been internally validated and is reliable for access. The IP verification encompassed all aspects, including connection integrity, block functionalities, and adherence to specifications.

The testing highlighted several areas of concern, particularly regarding partition violations, ID mismatches, and unauthorized access across multiple partitions within the IP. Despite initial challenges, continuous regression testing showed a progressive decrease in failing test cases and an increase in passing ones affirming thorough validation across every context. This ensures that IP is ready for deployment to the verification closure.

Chapter 5

Conclusion and Future Scope

Automation using ML finds many applications daily in wafer production and verification processes. For verification, it speeds up verification failure, increases the efficiency of verification, makes simulation more practical, improves integration, and makes debugging simulation faster and easier. This work plan enables debugging by dividing errors into boxes according to their characteristics. It also analyzes the raw regression parameters and identifies the root causes of errors in design. It reduces the time and improves the overall debugging approach with less effort. The implemented project gives coverage result analysis and ML model accuracy of 91.9% by covering aspects of register and functional test cases.

This automation technique which is scripting based debug optimization reduces the effort required to verify the complex designs. Considering the overall analysis, it can be concluded that debugging process is optimized for the verification process. Furthermore, there is potential for future enhancement by extending its application to other ML models which are decision based for comparative analysis. Introducing parameters specific to particular failures would enable the model to automatically analyze root causes. Additionally, enhancing the depth of epochs could render the process more dynamic. Furthermore, substituting deep learning models could potentially improve accuracy further. Reusability can also be added for the multiple function sequences. These steps collectively suggest avenues for advancing and refining the debugging framework to enhance its effectiveness in verifying complex systems.

Safety IP is verified at IP level with full code coverage and functional coverage. This IP is also verified for multiple interconnect scenarios of blocks at SoC level verification using C based test cases. It gives the overall coverage grade of 100% at SoC level, it verifies that all the NOC Connections are interlinked properly to other components. The VIPs environment of AXI and APB protocols have been developed and verified by the virtue of protocols. The code coverage proves the efficiency of the VIPs in providingthe path to achieve the coverage goal.

References

[1] A. Rezaei Aderiani, K. Wärmefjord, R. Söderberg, and L. Lindkvist, “Individualizing Locator Adjustments of Assembly Fixtures Using a Digital Twin,” *Journal of Computing and Information Science in Engineering*, vol. 19, no. 4, pp. 3-34, 2019.

[2] A. Wiemann, "Standardized Functional Verification," *1st ed. New York, NY: Springer US*, pp. 1–8, 2008.

[3] Ahmed Wahba, Justin Hohnerlein, Farhan Rahman, “Functional and Formal Verification based automated testbench*,” International Journal on Microprocessor/SoC Test, Security and Verification (MTV)*, 2021, pp.1222 -1267, 2020.

[4] Ali, A.M. Shalaby, A. Saif, "A UVM-based Verification Approach for MIPI DSI Low-Level Protocol layer*," International Journal of Microelectronics (ICM),*Casablanca, Morocco, pp. 74–77, 4–7 December 2022.

[5] Amr Hany, Ahmed Ismail, Ahmed Kamal, Mohamed Badran, "Approach for a Unified Functional Verification Flow," *Design Verification Technology Mentor Graphics Egypt*, pp.267-289, Mar 1995.

[6] Ankitha and D. Aradhya, “A python based design verification methodology*,” Journal of University of Shanghai for Science and Technology*, vol. 23, pp. 1–11, 2021.

[7] Arpitha O, Naik Elizabeth Kuruvilla, Arunkumar P Chavan, "Integration and Verification of IP Cores on SoC*," IEEE Mysore Sub Section International Conference*, pp. 192-221, 2021.

[8] Ashima Kukkar, Rajni Mohana, Yugal Kumar, "Duplicate Bug Report Detection and Classification System Based on Deep Learning Technique," *11th International Conference in Computer Science*, Korea (MSICT), pp.3–7, 2020.

[9] Ashok B. Mehta, "Constrained random verification,” *ASIC/SoC Functional Design Verification," Boston MA: Springer US*, pp. 65–74, 2018.

[10] Binod Kumar, Kanad Basu, Ankit Jindal, Masahiro Fujita and Virendra Singh, "Improved post-silicon error detection by selecting the topology of signal lines,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24. pp 313–323, January 2016.

[11] Brian Keng and Andreas Veneri, "Path-Directed Abstraction and Refinement for SAT-Based Design Debugging," I*EEE Transactions on Integrated Circuits and Systems Computer-Aided Design, vol. 32*, pp 1609 - 1622, October 10, 2013.

[12] D. Craigen, S. L. Gerhart, and T. Ralston, “Formal Methods Reality Check: Industrial Usage,” *IEEE Transactions on Software Engineering*, vol. 21, pp. 90-98, 1995.

[13] Daeseo Cha, Soonoh Kwon, and Ahhyung Shin, Youngnam Youn, Youngsik Kim, Seonil Brian Choi, "Metadata based Testbench Generation Automation," *IEEE Standard for SystemVerilog Unified Hardware Design, Specification, Vol.44, pp 113–167*, 2018.

[14] Djordje Maksimovic, Andreas Veneris, Zissis Poulos, "Clustering based Revision Debug in Regression Verification," *33rd IEEE International Conference on Computer Design*, pp. 1341- 1458, 2015.

[15] E. Mankolli and V. Guliashki, "Machine learning and natural language processing: Review of models and optimization problems" in ICT Innovations 2020. Machine Learning and Applications, Cham, Switzerland:Springer, pp. 71-86, 2020.

[16] El-Ashry, S. Adel, “Efficient Methodology of Sampling UVM RAL during Simulation for SoC Functional Coverage,” *19th International Journal on Microprocessor and SOC Test and Verification (MTV),* Austin, TX, USA, pp. 61–66, 9–10 December 2018.

[17] Eman El Mandouh and Amr G. Wassal, "Application of Machine Learning Techniques in Post-Silicon Debugging and Bug Localization," *Journal of Electronic Testing* Vol. 34, pp. 163–181, 2018.

[18] Eman El Mandouh, Laila Maher, Moutaz Ahmed, Yasmin ElSharnoby “Guiding Functional Verification Regression Analysis Using Machine Learning and Big Data Methods," *Design and Verification Conference and exhibition, Europe*, pp. 29-35, 2018.

[19] Ganapathy Parthasarathy, Aabid Rushdi, Parivesh Choudhary, Saurav Nanda, "RTL Regression Testing Selection Using Machine Learning," *27th Asia South Pacific Design Automation Conference*, 2022.

[20] Gaurav Sharma, Lava Bhargava & Vinod Kumar, "Automated Bug Resistant Test Intent with Register Header Database for Optimized Verification," *Journal of Electronic Testing Vol.36*, pp 219– 237, 2021.

[21] Gopal Sharma, Lava Bhargava, Vinod Kumar, "Automated Coverage Register Access Technology on UVM Framework for Advanced Verification," *IEEE International Symposium on Circuits and Systems (ISCAS)*, Vol.42, pp 198–235, 2018.

[22] Goldie A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and evaluation methods using Boolean satisfaction," *IEEE Trans. on CAD* pp. 378 -389, 2005.

[23] Harshitha, N.B. Kumar, Y.G.P. Kurian, "An Introduction to Universal Verification Methodology for the digital design of Integrated circuits," *International Conference on AI and Smart Systems (ICAIS),* Tamil Nadu, India, pp. 1710–1713, 25–27 March 2021.

[24] Hussien, A. Mohamed, S. Soliman, M. Mostafa, "Development of a Generic and a Reconfigurable UVM-Based Verification Environment for SoC Buses," *31st International Conference on Microelectronics (ICM),* Cairo, Egypt, pp. 195–198, 15–18 December 2019.

[25] J. R. Goldberg, "Design Verification and Validation," *Capstone Design Courses in Springer Nature Switzerland* AG, 2012.

[26] Joon-Sung Yang and Nur A. Touba, "Selection of signal indicators for testing high-performance debugging," *27th IEEE VLSI Testing Symposium*, 2009.

[27] Liang K. Wu, J. Ren, H. Zhang, "Design and Implementation of DSP Cache," *IEEE 21st International Conference on Communication Technology (ICCT),* Montreal, QC, Canada, pp. 993–997, 14–23, June 2021.

[28] Mustafa Efendioglu, Alper Sen, Yavuz koroglu, "Bug Prediction of SystemC Models Using Machine Learning," *IEEE Transactions on computer-aided design of integrated circuits and systems*, vol. 38, pp. 421-422, March 2019.

[29] Priyanshi Gaur, Sidhartha Sankar R and Sujay Deb, "Efficient Hardware Verification using Machine Learning Approach," *IEEE International Symposium on Smart Electronic System*, pp. 32-37, 2019.

[30] R. Cheng, "Revision debug in regression for clustering testcases," Computer Design (ICCD), *33rd IEEE International Conference*, pp. 33-36, Oct 2015.

[31] R. W. Brennan, L. Ferrarini, J. L. M. Lastra, and D. V. Vyatkin, "Automation objects: Enabling embedded intelligence in real-time mechatronic systems," *International Journal of Manufacturing and Resources*, vol. 1, pp. 379–381, 2006.

[32] S. Safarpour, A. Veneris, and R. Drechsler, "Improved SAT-based reachability analysis with observability don't cares," *International Journal on Satisfiability Boolean Modeling and Computation*, vol. 5, pp. 1–25, 2008.

[33] Samhita Varambally B, "Advancing Research on Design Using Machine Learning: An Open Source Program," *ASIC/SoC Functional Design Verification. Springer*, Vol. 26, pp. 1143-1221, 2018.

[34] Sebastian Siegfried Prebeck, Zhao Han, Deyan Wang, Gabriel Rutsch, Bowen Li, "Aspect-Oriented Design Automation with Model Transformation," *IEEE 29th International Conference on Very Large Scale Integration (VLSI-SoC)*, pp.256 - 278, 2021.

[35] Sherif Hosny, "Unified UVM Methodology for MPSoC Hardware/Software Functional Verification," *11th International Conference on Modern Circuits and Systems Technologies (MOCAST),* Bremen, Germany, pp. 1–5, 8–10 June 2022.

[36] Srinivasan, A. "A Short Look Back and a Longer Look Forward," *Journal of Machine Learning Research* 4, pp. 415–430, 2003.

[37] Varghese M P, T.Muthumanickam, "Machine Learning Approach for Electronic Design Automation in IC Design Flow*," 6th International ISMAC Conference*, pp.531-534, 2022.

[38] Wang, J. Tan, N. Zhou, Yangfan Zhou, Ting Li, "UVM Verification Platform for RISC-V SoC from Module to System Level," *IEEE 5th International Conference on Integrated Circuits and Microsystems (ICICM),* Nanjing, China, pp. 242–246, 23–25 October 2020.

[39] Yugeshwari Nemade, Prathmesh Pawar, Pranjali Kadhi, Rohini Shingne, Ujwala Ghodeswar, "UVM based Design Verification with Semi-Automation*," International Conference on Pervasive Computing and Social Networking, Abu Dhabi, United Arab Emirates*, pp.1120-1130, 2023.

[40] Yugeshwari Nemade, Prathmesh Pawar; Pranjali Kadhi, Rohini Shingne, "UVM based Design Verification with Semi Automation" *3rd International Conference on Pervasive Computing and Social Networking*, pp. 1622-1980, 2023.

[41] Yu-Shen Yang, Andreas Veneris Nicolla Nicolli, "Automating Data Analysis and Acquisition Setup in a Silicon Debug Environment," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, pp.1118-1129, June 2012.

[42] Zhou, S. Geng, S. Peng, X. Zhang, "The Design Of UVM Verification Platform Based on Data Comparison," *International Conference on Electronic Information Technology and Computer Engineering, Xiamen*, China, pp. 1080–1085, 22–24 October 2021.

Priyamj_602262014_thesis

ORIGINALITY REPORT

1%	0%	0%	1%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Thapar University, Patiala Student Paper	<1%
2	Gaurav Sharma, Lava Bhargava, Vinod Kumar. "Automated Bug Resistant Test Intent with Register Header Database for Optimized Verification", Journal of Electronic Testing, 2020 Publication	<1%
3	www.thinkmind.org Internet Source	<1%

Exclude quotes	On	Exclude matches	Off
Exclude bibliography	On		