Streamlining Design Verification and Protocol Validation

A Project submitted in partial fulfillment of the requirement for the

Award of the Degree of

MASTER OF TECHNOLOGY

in VLSI Design

Submitted By

Hritik Jaiswal

602362042

Under Supervision of

Dr. Sanjay Sharma & Dr. Arnab Pattanayak

Professor & Assistant Professor

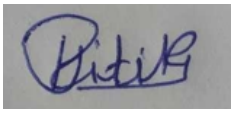
ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT
THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
(A DEEMED TO BE UNIVERSITY), PATIALA, PUNJAB

June 2025

DECLARATION

I, Hritik Jaiswal hereby declare that the work presented in this report entitled "**Streamlining Design Verification and Protocol Verification**" submitted at the **Department of Electronics and Communication Engineering, Thapar Institute of Engineering & Technology, Patiala** is an authentic record of work carried out under the supervision of **Dr. Sanjay Sharma**(Professor) and **Dr. Arnab Pattnayak** , Electronics and Communication Department. Thapar Institute of Engineering & Technology, Patiala) and **Mr. Sachin Jain** (Director Hardware Design, Intel India Pvt. Limited) from Jun 2024 to June 2025. The matter presented in this has not been submitted either in part or full to any other university or institute.

Date: 14 June 2025



Hritik Jaiswal

(Mr. Sachin Kumar Jain)  Director Hardware Design Intel India Pvt. Ltd Date: 14 June 2025	(Dr. Sanjay Sharma) Professor  Department of Electronics and Communication Engineering, Thapar Institute of Engineering & Technology, Patiala Date: 14 June 2025
	 (Dr. Arnab Pattanayak) Assistant Professor Department of Electronics and Communication Engineering, Thapar Institute of Engineering & Technology, Patiala Date: 14 June 2025

INTEL INDIA PRIVATE LTD

No 23-56, SRR Campus, Outer Ring Road, Devarabeesanahalli, Varthur Hobli,
Bellandur,Bengaluru, Karnataka, 560103, India

CERTIFICATE

Regd. Office:
Intel Technology India Private Limited
23-56P, Outer Ring Road,
Devarabeesanahalli, Varthur Hobli website: www.intel.in
Bellandur Post
Bangalore 560 103, India
CIN-U85110KA1997PTC021606

Tel: +91-80-2605 3000
Fax: +91-80-2605 6190

To Whomsoever It May Concern

WWID: 12269980

Employee Name: HRITIK JAISWAL

Internship Dates: 24/06/2024 to 23/06/2025

The letter is to confirm the mentioned above has undergone internship at Intel Technology India Private Limited.

We wish you all the best for your future assignments.

Yours Sincerely

Gowre Saseedaran

Date: June 29, 2025

Place: **Bangalore**

Acknowledgement

The satisfaction and euphoria that accompany the successful completion of any task would be incomplete without mentioning the people who made it possible. This humble endeavor bears the imprint of many persons who were in one way or the other helped for the completion of this thesis. With deep gratitude, I acknowledge all those guidance and encouragement, which served as a beacon of light and crowned our efforts with success.

I would like to thank Dr. Kulbir Singh for giving such an opportunity of internship at Intel India Pvt. Ltd. extend my deepest gratitude to Mr. Sachin Jain, Director Hardware Design Intel India Pvt. Limited for the valuable support he provided me.

I owe my heartfelt gratitude to Dr. Sanjay Sharma and Dr Arnab Pattnayak for their continuous support, motivation, immense knowledge. His guidance has helped me in all the time of research and writing of the thesis.

My sincere thanks and gratitude to my mentor Mr. Sachin Jain who has always supported me in taking new initiatives, accomplishing goals and comprehensive learning. Thanks a lot, to my mentor for his valuable guidance during the report work, He has given me valuable advice and support which has helped me in understanding the technical aspects of the report.

Last but not the least I thank my family, friends, and colleagues for their timely help and valuable suggestions.

Hritik Jaiswal

ABSTRACT

Design verification is a critical phase in the development of Intellectual Property (IP) cores to ensure they function correctly and meet the intended specifications. As IP cores are increasingly reused across multiple systems, the need for a robust and efficient verification process has become more crucial. This paper explores various methodologies and tools used for IP verification, including simulation-based techniques, formal verification, and hardware-assisted approaches like FPGA prototyping. We highlight the importance of creating a comprehensive testbench, using coverage metrics, and adopting universal verification methodologies (UVM) to improve verification efficiency and quality. The challenges posed by the growing complexity of IP cores and the need for scalability in verification are also discussed. The ultimate goal is to achieve a bugfree design before integration into larger systems, reducing time-to-market and ensuring reliable performance in end products.

By addressing the evolving challenges of IP verification, this paper provides insights into best practices and emerging trends in the field, supporting more efficient design cycles and higher quality IP cores.

TABLE OF CONTENT

List of Figures

List of Tables

Chapter 1: Introduction

Design verification is essential in the lifecycle of Intellectual Property (IP) cores, ensuring they function as intended and meet specifications. As the demand for reusable and reliable IP cores grows, an effective verification process becomes crucial to avoid costly errors and delays. This process directly impacts the quality, performance, and reliability of final products.

Various verification methodologies, including simulation, formal verification, and hardware-assisted techniques, are employed to validate IP functionality and performance. Metrics to measure verification success, such as test coverage and completeness, help assess how well the design has been tested and where improvements are needed.

As IP cores become more complex, scalable and efficient verification strategies are necessary to address challenges in verification. Emerging trends, such as automation, machine learning, and AI-based tools, are enhancing the verification process, increasing efficiency, and reducing human error.

A key tool in this process is System Verilog Assertions (SVA), which formally checks design properties, providing additional assurance that designs behave as expected. These advancements in verification methods ensure the successful development of high-quality IP cores, making verification a critical aspect of modern electronic design.

In the modern semiconductor industry, Intellectual Property (**IP**) cores are fundamental building blocks used in the design of complex systems on chips (**SoCs**). These IP cores, which can represent anything from a simple peripheral to a complex processor, are frequently reused across different designs and product generations. As these IP cores are often designed by third parties or reused from previous projects, ensuring their correctness and compatibility with a new system requires a rigorous design verification process. [1]

Design verification is a critical step that ensures the IP core adheres to its specifications and operates as intended under all possible conditions. In fact, studies show that design errors identified late in the production cycle can lead to significant cost overruns, project delays, and reliability issues in final products, therefore, plays a pivotal role in detecting and resolving issues early in the design phase, reducing overall development costs and time-to-market pressures.

The design verification process typically involves creating a **verification plan** based on the IP core's specifications, followed by constructing a **testbench** to simulate the core's functionality under a variety of scenarios. Commonly, engineers use **simulation-based verification**, wherein the IP is tested under different input conditions to ensure proper functionality. However, with increasing IP complexity, verification engineers also employ **formal verification** techniques, which mathematically prove that certain properties of the design hold true across all possible input cases

Moreover, **hardware-gasification** methods, such as **FPGA prototyping** and **emulation**, have gained prominence. These methods provide faster simulation speeds and more comprehensive validation of the IP core’s performance in realworld scenarios. To streamline the verification process, handle the

increasing complexity of IP cores, **Universal Verification Methodology (UVM)** has become a widely adopted industry standard. UVM provides a scalable and reusable verification framework, enabling the development of sophisticated, reusable testbenches that can be applied across different projects

As the complexity of semiconductor devices continues to rise, the **design verification** process has evolved into one of the most resource-intensive and crucial aspects of IP core development. Verification typically consumes more than 50% of the overall development time, which underscores its significance in ensuring the quality and reliability of modern integrated circuits (ICs). Inadequate verification can result in defects that may only surface after the product is fabricated or, worse, after it reaches the end user, leading to costly recalls or system failures.

1.1 Role of Design Verification in the IP Lifecycle

Verification is integral at every stage of the IP core lifecycle, from early development through final integration. In the initial stages of IP development, the verification process ensures that the design matches the functional specifications. As IP cores are refined and reused across multiple projects, the verification process helps ensure compatibility with various configurations and system architectures. This is especially important as modern **SoCs integrate multiple IP**

blocks, each with potentially different communication protocols, clock domains, and power management schemes. A primary objective of design verification is to identify functional bugs and corner cases, which are rare but critical scenarios that may not be exercised during normal operation. These corner cases can lead to system failure if not properly accounted for in the design process. By systematically verifying the design under a broad range of conditions, engineers can catch and correct errors early, minimizing the risk of failure. [2]

Table 1.1 Acronyms Definition

ABV	Assertion-Based Verification
CDV	Coverage-Driven Verification
PSS	Portable Stimulus Standard
RTL	Register Transfer Level
UNR	Unreachable Code Report
FSM	Finite State Machine
NLP	Natural Language Processing
LTL	Linear Temporal Logic
CTL	Computation Tree Logic
NuSMV	New Symbolic Model Verifier

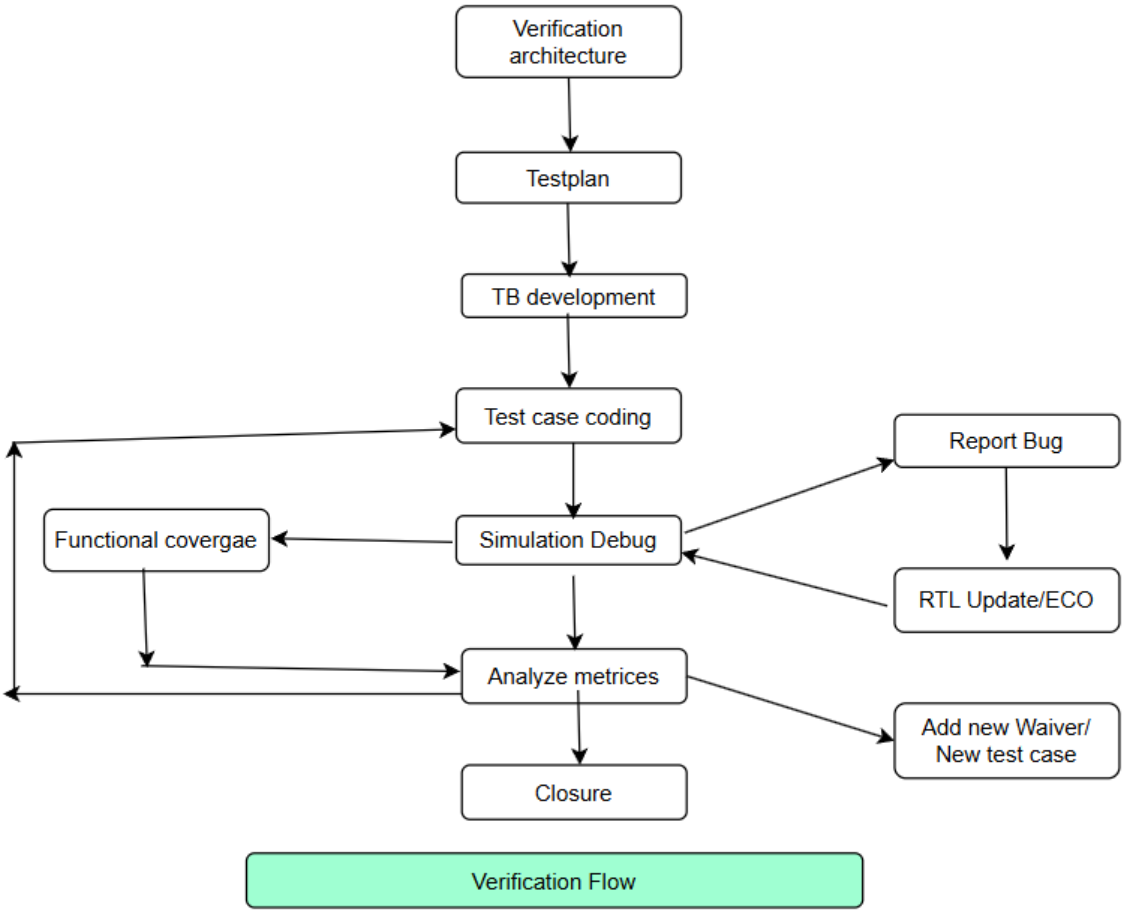


Figure 1.1 ASIC Verification flow

1.2 Common Verification Methodologies

Several methodologies are used to achieve thorough verification of IP cores as shown in figure 1.1, each offering unique advantages:

Simulation-Based Verification:

This is the most common approach, where the IP design is simulated under various input stimuli. Engineers develop testbenches, which generate stimuli and monitor outputs, to compare against expected behaviours. Modern testbenches are often written in **high-level hardware description languages (HDLs)** such as **SystemVerilog**. Simulation tools from vendors like **Cadence**, **Synopsys**, and **Mentor Graphics** are widely used in this process. Despite its effectiveness, simulation can be time consuming for large designs, especially when trying to achieve high coverage of possible states.[3]

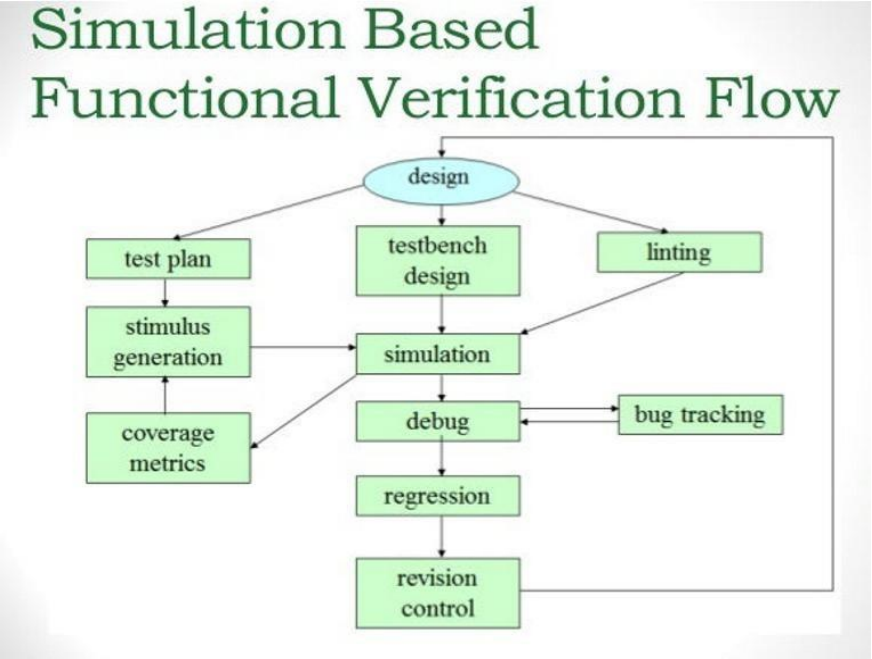


Figure 1.2 Functional Verification Flow [4]

Figure 1.2 represents the **Simulation-Based Functional Verification Flow**, which outlines the steps and processes involved in verifying a hardware design through simulation.

i. Design

- The hardware design (DUT - Design Under Test) is the focus of the verification process.
- This is typically described using an HDL like Verilog or VHDL.

ii. Test Plan

- Defines the objectives of the verification, including:
- Scenarios to be tested.
- Features to be covered.
- Metrics to measure success.
- A well-defined test plan ensures structured and complete verification.

iii. Testbench Design

- The testbench is an environment created to stimulate the DUT and verify its outputs.
- It includes stimulus generators, monitors, checkers, and scoreboards.

iv. Linting

- A static analysis step to catch coding errors, bad practices, or inconsistencies in the design before simulation.
- Ensures clean HDL code, reducing potential bugs in later stages.

v. Stimulus Generation

- Input stimuli are created to exercise the DUT.

. Can be **directed tests** (manually crafted scenarios) or **random tests** (automated test generation).

- Stimulus tests edge cases, boundary conditions, and normal operations.

vi. Simulation

- The DUT and testbench are run on a simulator (e.g., VCS, ModelSim).
- Simulation produces output waveforms and logs for analysis.

vii. Coverage Metrics

- Measures how thoroughly the design has been tested.
- Examples: Code coverage, functional coverage, toggle coverage.
- Ensures no critical part of the design is left unverified.

viii. Debug

- If the DUT's output doesn't match expected results, debug logs and waveforms are analysed to find and fix issues.
- May involve modifying the DUT, testbench, or stimulus.

ix. Bug Tracking

• Bugs identified during debugging are logged, categorized, and tracked for resolution. • This ensures systematic handling of all issues.

x. Regression

- Regression testing involves running all previously created tests to ensure new changes haven't introduced bugs.
- It ensures consistency and prevents regressions in functionality.

xi. Revision Control

- Manages changes to the design, testbench, and test cases using tools like Git.
- Helps in collaboration, tracking history, and rollback if needed. [3]
- **Formal Verification**: Unlike simulation, which tests the design under selected inputs, formal verification uses mathematical techniques to prove the correctness of a design for all possible inputs and states. This method is particularly useful for checking specific properties, such as ensuring that deadlock cannot occur or that certain safety conditions are always met. Formal verification is limited by scalability issues, but it has proven effective for verifying complex control logic and ensuring compliance with security protocols. [4]
  -

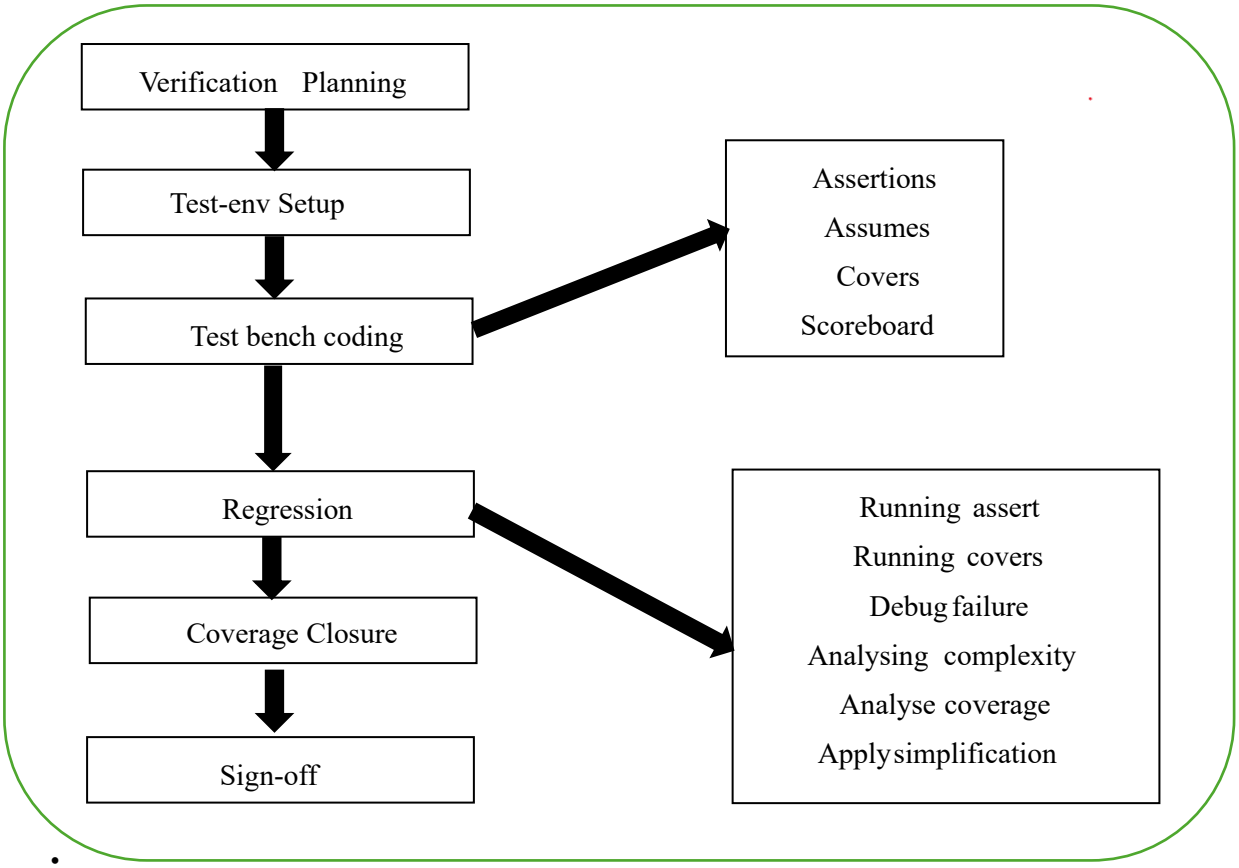


Figure 1.3 Formal Verification Life Cycle[4,5]

This diagram in figure 1.3 represents the **Formal Verification Life Cycle**, which is used for ensuring correctness in hardware designs through mathematical proofs rather than simulation.[4]

i. Verification Planning

- **Objective**: Define the properties and constraints for verification.
- **Activities**:
- Identify key behaviours or properties of the design to verify.
- Create a plan for writing assertions, covers, and assumptions.

ii. Test Environment Setup

- **Objective**: Set up the environment for formal tools.
- **Components**:
- **Assertions**: Specify the properties that the design must satisfy.
- **Covers**: Track if specific conditions or states are reached.
- **Scoreboard**: Collect and compare verification results.

- **Assumes**: Specify conditions assumed to hold during verification.

iii. Test Bench Coding

- **Objective**: Code the necessary assertions, assumptions, and covers.
- **Activities**:
- Write properties for critical design functions.
- Implement monitors and verification constraints.
- Ensure the environment aligns with the formal verification tool.

iv. Regression

- **Objective**: Run the formal tool iteratively to explore all possible scenarios.
- **Activities**:
- **Run Assertions**: Check for property violations.
- **Run Covers**: Ensure intended design states are reached.
- **Debug Failures**: Analyse counterexamples when properties fail.
- **Analyse Complexity**: Handle state-space explosion in complex designs.

v. Coverage Closure

- **Objective**: Ensure all specified properties and conditions are verified.
- **Activities**:
- Analyse which properties or covers are incomplete.
- Apply optimizations or simplifications to improve tool performance.
- Add more assertions or covers if necessary. [5,6]

vi. Sign-Off

- **Objective**: Finalize verification with proofs of correctness.
- **Activities**:
- Ensure all critical properties are formally proven.

- Validate that all coverage goals are met.
- Approve the design for the next phase.

- **Hardware-Assisted Verification**: To accelerate verification, **FPGA based prototyping** and **hardware emulation** are often employed. These techniques provide near-real-time validation by mapping the IP design onto an FPGA or emulation platform, enabling the verification of large designs that would be too slow to simulate. By running real software or systemlevel tests on the hardware prototype, verification engineers can catch errors that would be difficult to uncover through simulation alone.

1.3 Metrics for Verification Success

To quantify the thoroughness of the verification process, various metrics are employed:

- **Code Coverage**: This measures how much of the design's code has been exercised during simulation. It includes statement, **branch**, and **condition** coverage, indicating the extent to which different parts of the design have been verified.

- **Functional Coverage**: Functional coverage ensures that specific functionality described in the verification plan is tested. For instance, if an IP block contains a **state machine**, functional coverage would verify that all possible state transitions are exercised during simulation.

- **Assertion-Based Verification**: Assertions are properties that must hold true at certain points in the design. Assertion-based verification (ABV) helps catch subtle bugs by embedding checks within the design, which are triggered when certain conditions fail during simulation.

1.4 Challenges in IP Verification

As IP cores grow in complexity, verification engineers face several challenges. One key challenge is achieving **scalability**. Modern IP designs are highly configurable, with multiple operating modes, power states, and data paths. Verifying all combinations of these configurations is infeasible using traditional methods alone, prompting the need for sophisticated verification strategies, including **constrained-random testing** and **coverage-driven verification**. [3]

Another challenge is ensuring **reuse of verification components**. IP cores are often reused in multiple projects, and verification environments must be modular and reusable across different configurations. This is where **Universal Verification Methodology (UVM)** comes into play, offering a structured and

reusable framework for developing testbenches and ensuring that verification efforts scale with the increasing complexity of designs.

Cache coherency protocols, such as **MESI and MOESI**, add additional complexity. Verifying correct state transitions, ordering, and data consistency across private caches in a multi-core system requires sophisticated testbenches and coverage models.

1.5 Emerging Trends in IP Verification

As design complexity continues to grow, new verification methodologies are emerging:

Machine Learning (ML) in Verification: Machine learning algorithms are being applied to optimize test generation, identify critical corner cases, and prioritize coverage holes. ML-based tools can analyse past simulation results and dynamically adjust verification strategies to focus on undertested portions of the design.

Portable Stimulus Standard (PSS): PSS is a specification-driven approach that allows for the reuse of test scenarios across different levels of abstraction and platforms. This is particularly useful for verifying IP cores in various environments, including simulation, emulation, and postsilicon testing.

Security and Safety Verification: With the growing importance of security in connected devices, verifying that IP cores are free from vulnerabilities, such as timing attacks or side channel leakage, is becoming critical. Safety-critical applications, such as automotive and aerospace systems, require additional verification steps to ensure compliance with standards like ISO 26262 for functional safety. [7,8]

1.6 Overview of Fabric and Interface Protocols

A **fabric** in an SoC is the interconnect mechanism that enables communication between different IP blocks. It acts as a data highway, allowing multiple subsystems (processors, memory controllers, peripherals) to exchange information efficiently.

An **interface protocol** defines the set of rules governing how IP blocks communicate over the fabric. These protocols standardize signal transactions, data formats, arbitration mechanisms, and handshaking procedures, ensuring seamless communication across heterogeneous components.

1.7 Types of Interface Protocols

Interface protocols can be categorized into standard and custom protocols, each suited for specific SoC architectures and applications.

Standard Protocols

Standardized interface protocols provide a widely accepted framework for SoC communication. Some commonly used protocols include:

- AMBA (Advanced Microcontroller Bus Architecture) – Developed by ARM, widely used in modern SoCs.

- AXI (Advanced eXtensible Interface) – High-performance, high-bandwidth interface for processor-memory communication.
- AHB (Advanced High-performance Bus) – Single-bus protocol used for moderate-speed peripherals.
- APB (Advanced Peripheral Bus) – Low-speed, low-power protocol for peripheral devices.

- PCIe (Peripheral Component Interconnect Express) – High-speed serial interface for interconnecting high-performance components like GPUs and network controllers.
- USB, I2C, SPI – Serial communication protocols used for external device connectivity.

1.8 AMBA APB Protocol

Advanced Peripheral Bus (APB) is a low-power, low-bandwidth interface within the Advanced Microcontroller Bus Architecture (AMBA), developed by ARM. It is primarily designed for connecting simple peripheral devices in System-on-Chip (SoC) designs. Unlike the high-performance Advanced eXtensible Interface (AXI) and Advanced High-performance Bus (AHB), which are optimized for high-speed data transfers, APB is optimized for simplicity and power efficiency, making it ideal for peripheral devices that do not require high-speed data movement.

APB is commonly used for interfacing with peripherals such as General-Purpose Input/Output (GPIO), Universal Asynchronous Receiver/Transmitter (UART), timers, interrupt controllers, and watchdog timers. These peripherals often operate at lower data rates and do not require complex transaction mechanisms like burst transfers or pipelining.

Key Characteristics of AMBA APB

Simplicity – Uses a non-pipelined, single-cycle transfer approach for easy implementation.

Low Power Consumption – Ideal for battery-powered and low-power SoC designs.

Reduced Area Overhead – Requires fewer logic gates, making it space-efficient.

Easy Integration – Designed to be a bridge between high-speed buses (AXI/AHB) and slow peripherals.

Non-Pipelined Transfers – Unlike AXI and AHB, APB executes transactions in a sequential manner, which reduces complexity.

Role of APB in SoC Architectures

In a typical SoC, multiple components interact with each other via an interconnect fabric that enables communication. This interconnect consists of high-performance buses (AXI or AHB) for processing elements and memory, along with APB for connecting slower peripherals. The AHB-to-APB bridge plays a crucial role in linking APB peripherals to the main system bus, ensuring smooth data transfers between different parts of the SoC.

APB in AMBA Hierarchy

AMBA defines a hierarchical bus structure:

AXI (Advanced eXtensible Interface) – High-performance, high-bandwidth interface used for memory and processor communication.

AHB (Advanced High-performance Bus) – Used for high-speed peripherals like DMA controllers, high-speed memory interfaces, etc.

APB (Advanced Peripheral Bus) – Used for low-speed peripherals such as timers, UARTs, GPIOs, and control registers.

APB Protocol Features

1. Simple Bus Architecture

Unlike AXI and AHB, which use burst transfers and pipelined operations, APB follows a single-cycle transfer mechanism where each transaction completes in a single clock cycle (excluding setup time). This makes it easy to implement and power-efficient.

2. Master-Slave Communication

APB follows a single-master, multiple-slave architecture, where a single master (typically an AHB-to-APB bridge) communicates with multiple slave peripherals. The master initiates the transactions, while the slave devices respond to the requests.

3. Sequential Transfer Mechanism

APB does not support burst transfers, meaning each transaction occurs one at a time.

It follows a request-acknowledge protocol, where the master waits for a response from the slave before initiating the next transfer.

4. Low-Power Operation

Since APB operates only when needed (rather than running continuously like AXI/AHB), it reduces unnecessary power consumption.

The bus remains idle when no transactions are occurring, further optimizing power efficiency.

5. Address and Data Phases

Each APB transfer consists of two key phases:

Address Phase – The master sends the address and control signals to the slave.

Data Phase – The slave responds by sending or receiving data as per the operation (read/write).

APB Transaction Mechanism

A typical APB transaction follows these steps as shown in table 1.2 and figure 1.5 :

Idle Phase:

PENABLE = 0, Psel=0

Setup Phase:

The master sets up the address, write/read signal, and enable signal (PENABLE = 0).

The slave prepares to receive the request.

Access Phase:

The master asserts PENABLE = 1, indicating that the transfer is active.

The slave reads the address and processes the request.

Completion:

The slave acknowledges the request by providing data (in case of a read operation) or storing the data (in case of a write operation).

The master deactivates the PENABLE signal, and the transaction ends.

Timing Diagram for APB Transactions

APB transactions consist of a single address phase followed by a single data phase. Here's how it works:

PSEL (Peripheral Select) – Indicates which peripheral is being accessed.

PWRITE – Determines if the transaction is a write (1) or read (0) operation.

PENABLE – Enables data transfer during the access phase.

PRDATA/ PWDATA – Data transferred from the slave (read) or to the slave (write).

Table 1.2 State table of APB Master[9]

Cycle	PSEL	PENABLE	PWRITE	PADDR	PWDATA/PRDATA	Description
1	1	0	1	Address	Data	Setup phase
2	1	1	1	Address	Data	Access phase
3	0	0	-	-	-	Transaction complete

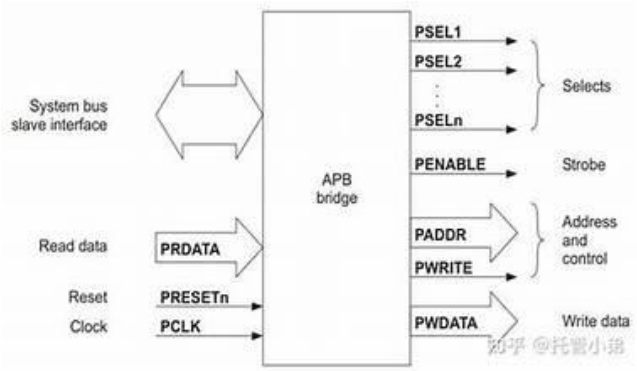


Figure 1.4 APB Bridge[9]

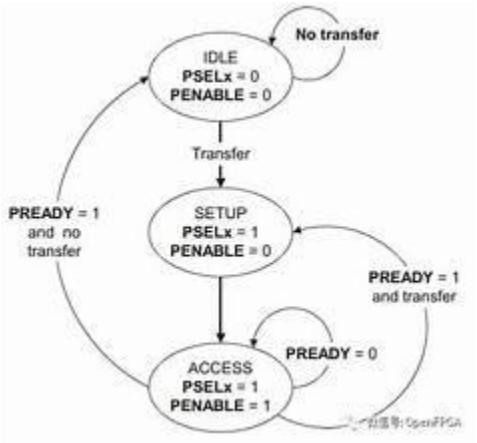


Figure 1.5 State diagram for APB Master[10]

Chapter 2: Literature Survey

Covergroups are fundamental in functional coverage, allowing verification engineers to measure and ensure the thoroughness of tests against a design's specification. As designs grow increasingly complex, manual creation and management of covergroups become time-consuming and prone to human error. Automating covergroups generation and analysis has become a critical area of research to streamline verification efforts and improve efficiency. This literature survey reviews key contributions and methodologies for automating covergroup definition, optimization, and management in SystemVerilog verification environments.

2.1 Constraint-Based Automation

Constraint-based methods for automated covergroup generation focus on capturing critical functional scenarios and coverage points without manual intervention. Researchers have proposed various algorithms that analyse the design and automatically generate covergroups based on the constraints and test scenarios defined in the verification plan.

For example, in **Cheng et al. (2017)**, a tool was developed to automatically extract coverage points from design specifications using formal methods. This automated approach reduces the time spent manually defining covergroups and increases coverage precision, especially for complex control logic and state machines. [2]

Another relevant work by **Kumar et al. (2019)** introduces a method to use highlevel design specifications to generate covergroups based on constrained-random verification inputs. The system automatically identifies key inputs and output dependencies, generating coverpoints that target critical paths and corner cases of the design.

2.2 AI and Machine Learning in Covergroup Automation

Machine learning (ML) techniques have started to play a pivotal role in optimizing the coverage process. ML algorithms analyse coverage reports and simulation data to detect gaps in coverage and suggest new coverpoints. In **Wang et al. (2020)**, an ML-based tool was developed to automatically refine and generate new covergroups based on historical simulation data. The tool applies unsupervised learning to detect untested scenarios that may have been missed by manual analysis, leading to an adaptive verification process[5].

Additionally, **Gupta et al. (2021)** proposed using reinforcement learning to dynamically adjust the covergroup definition during simulation. Their technique enables the verification environment to automatically expand, or contract coverage targets based on real-time simulation results, focusing on hard-to-reach or under-tested conditions in the design. [6]

2.3 Coverage Closure Optimization

Coverage closure refers to the point when no additional meaningful coverage can be achieved without hitting diminishing returns. In their study, **Singh et al. (2018)** introduced an automated method for optimizing covergroup bins by removing redundancies and focusing only on coverpoints that contribute to coverage goals. Their algorithm identifies overlapping covergroups and merges them to reduce simulation complexity while maintaining coverage accuracy.

2.4 Cross Coverage Optimization

Cross coverage is essential for ensuring that multiple signals or parameters are exercised together in meaningful combinations. However, manually defining cross coverage for every possible combination can be inefficient. **Roy et al. (2019)** proposed a heuristic-based optimization technique that automatically identifies critical signal interactions and generates only the most relevant cross coverage pairs. This reduced the total number of cross combinations without sacrificing the quality of functional coverage.

2.5 Universal Verification Methodology (UVM)

A typical UVM testbench consists of several key components shown in 2.1, each with a well-defined role in the **verification flow**:

Sequence and Sequence Item: Sequences are the source of stimulus in a UVM testbench. A sequence generates a stream of **sequence items** (transactions) that represent abstract protocol-level operations. For example, in an APB testbench, a sequence item might represent a read or write transaction.

Sequencer: The sequencer controls the ordering and flow of sequence items to the driver. It provides arbitration and synchronization services for sequences running on the testbench.

Driver: The driver receives sequence items from the sequencer and translates them into **pin-level activity** or signal-level transactions that are driven onto the **Design Under Test (DUT)** interface.

Monitor: The monitor observes the DUT's interface signals passively and reconstructs high-level transactions. It forwards these transactions to other components such as the scoreboard and coverage collector.

Scoreboard: The scoreboard performs checking by comparing actual DUT behavior against expected results. It validates data integrity, ordering, and protocol compliance.

Agent: An agent is a hierarchical container that encapsulates the sequencer, driver, and monitor for a given interface or protocol. Agents can operate in **active mode** (driving stimulus) or **passive mode** (observing DUT behavior).

Environment (env): The environment aggregates one or more agents, the scoreboard, and any additional verification components. It provides a top-level container for the entire verification structure.

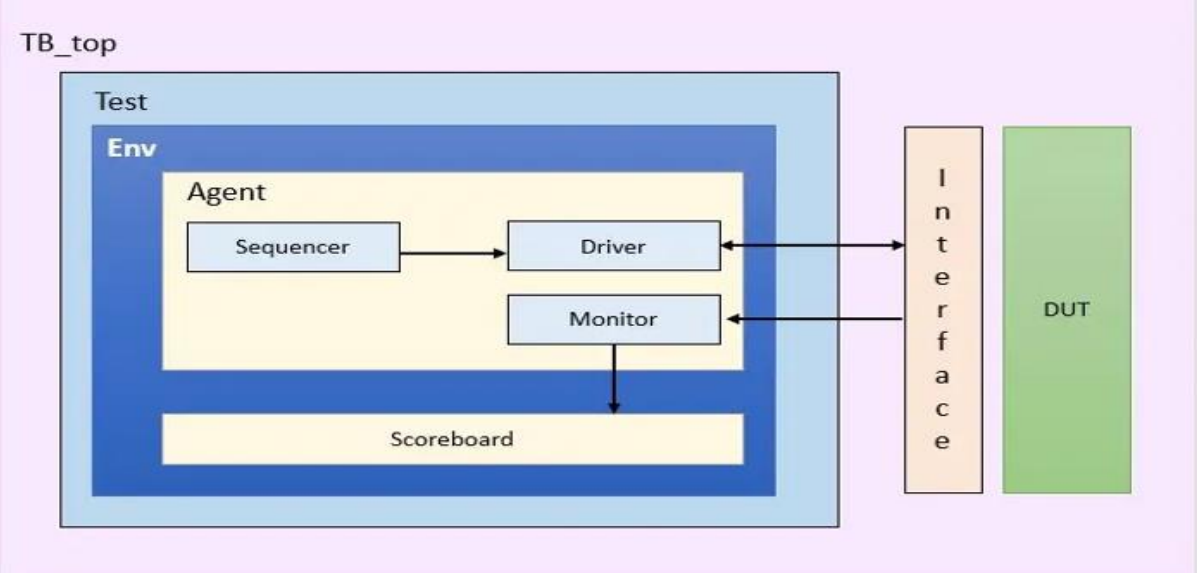


Figure 2.1 UVM Flow[11]

The flow of transactions in a UVM testbench typically proceeds as follows: The **UVM test** starts execution and configures the environment. The environment builds the necessary agents and scoreboards. The test then initiates one or more **sequences** on the appropriate sequencers. The **sequencer** sends **sequence items** to the **driver**, which drives them onto the DUT interface.At the same time, the **monitor** passively observes the DUT's interface signals and reconstructs transactions. These transactions are forwarded to the **scoreboard** for checking. Additionally, coverage models can be updated to track functional coverage.

Test: The UVM test is the top-level component that configures and controls the environment. It can specify which sequences to run, how agents are configured, and what coverage goals are targeted.

2.6 AMBA APB Protocol and SoC Interconnects

P. Mishra et al. (2014) compared APB with AXI and AHB in terms of latency, throughput, and power consumption. Their results showed that while APB had higher latency due to its non-pipelined structure, it consumed 50% less power than AXI when connecting simple peripherals.[9]

Key Findings

- APB has **higher latency** but is **more power-efficient** than AXI and AHB.
- Ideal for **control and low-speed peripheral communication** in SoCs.
- FPGA implementations confirm its **hardware simplicity and low resource utilization**.

2.7 Performance Analysis of APB in SoCs

G. Kumar et al. (2017) proposed a **UVM-based verification** methodology for AMBA APB to ensure proper bus functionality in SoC environments. Their approach successfully detected common bus transaction errors, improving design reliability.[11]

Key Findings

- **UVM-based verification** ensures APB protocol compliance in SoCs.
- **Clock synchronization issues** must be managed when interfacing APB with high-speed buses.
- **Assertion-based verification (ABV)** helps detect protocol violations early in the design phase.

Chapter 3: Research Gaps

IP (Intellectual Property) verification is a crucial phase in the design and development of integrated circuits (ICs), ensuring that individual IP cores function as expected before they are integrated into larger systems. Despite significant advancements in verification methodologies, several **research gaps** remain, particularly as designs grow in complexity, configurability, and size. Identifying and addressing these gaps is critical for improving verification efficiency, reducing time-to-market, and enhancing overall design quality. The following sections outline key areas where gaps persist in IP verification.

3. 1 Automated Coverage for Highly Parameterized Designs

As IP designs become more modular and parameterized, especially in SoCs (System-on-Chips), existing tools often struggle to automatically generate comprehensive and reusable functional coverage models that scale with different configurations. Dynamic covergroup generation methods are still in their early stages, and more research is needed to address how to cover all possible configurations effectively. [6]

3. 2 Machine Learning for Coverage Optimization

Machine learning (ML) techniques are beginning to be applied to coverage optimization, but these methods are still under-researched. ML could be leveraged to identify optimal coverage points dynamically and adapt verification strategies, but existing tools do not fully explore the potential of AI for this purpose. Further work is needed to integrate AI/ML into automated coverage feedback loops. [5]

3. 3 Verification of Power and Performance Trade-off

While power verification tools exist, the verification of power-management features (like **dynamic voltage scaling or clock gating**) within IPs is still not well-integrated into standard functional verification flows. This gap is particularly problematic for low-power designs, where power states must be thoroughly tested alongside functionality. Research into unified power-aware functional verification methodologies is limited.

3. 4 Performance Verification Under Dynamic Power Conditions

Verifying performance constraints, such as timing and throughput, in environments where power management schemes are active remains a challenge. Current tools often decouple functional verification from power and performance constraints, making it hard to capture corner cases where low power modes could degrade performance or cause failures.

3. 4 Performance Optimization for Low-Latency Applications

- APB's **single-cycle, non-pipelined architecture** leads to **higher transaction latency**, making it unsuitable for time-sensitive applications.
- Lack of **burst transfer support** limits its use in scenarios where multiple peripheral accesses are required in quick succession.

3. 4 Power Efficiency Enhancements

- While APB is inherently low-power, existing implementations do not fully exploit power-saving techniques like adaptive clock scaling, power gating, or dynamic voltage adjustments.
- Current APB4 write strobe signals provide some power savings, but further dynamic power management techniques are underexplored.

Chapter 4: Problem Statement based on Research Gaps

4. 1 Enhancing Verification Efficiency through Automated System Verilog Coverage

Automated System Verilog coverage boosts verification efficiency by reducing manual effort and increasing the thoroughness of design testing. System Verilog supports two main types of coverage. Automation tools streamline the generation of coverpoints and covergroups, minimizing errors and ensuring complete coverage. This approach is integral to coverage-driven verification (CDV), allowing verification engineers to focus on refining test scenarios, improving overall test quality, and reducing the time to verification closure**.** [12,13,14,15]

4. 2 Simplifying Regression Analysis: Automating Report Generation with Python

Automating regression analysis report generation with Python simplifies data analysis by streamlining repetitive tasks and ensuring consistency. Python libraries like pandas, statsmodels, and matplotlib enable users to automate data preprocessing, model fitting, and visualization. By scripting the entire workflow, analysts can efficiently produce standardized, detailed reports with key metrics, statistical summaries, and graphs. This automation saves time, reduces errors, and enhances the reproducibility of analysis, making it a valuable tool for both data scientists and business analysts.[14,15,16,17]

4. 3 Resolving Test Failures: Primary Debugging Techniques in Design Verification

In design verification, resolving test failures involves key debugging techniques:

i. Waveform Analysis: Examining signal transitions to identify mismatches between expected and actual behaviour.

ii. Assertion-Based Debugging: Using assertions to catch violations of design properties and narrow down issues.

iii. Log Analysis: Reviewing simulation logs for error messages or unexpected behaviour.

iv. Comparing Golden Models: Checking discrepancies between the design under test (DUT) and a reference model.

v. Binary Search on Testbenches: Narrowing down failure points by isolating changes that lead to errors.[16,17,18,19]

4. 4 Enhancing Handshake Protocol Verification Through System Verilog Assertions

System Verilog Assertions (SVAs) significantly improve handshake protocol verification by automating the detection of protocol violations. Key benefits include:

i. **Real-Time Monitoring**:

SVAs check protocol correctness during simulation, verifying that signals like "request" and "acknowledge" follow the expected timing and sequence.

ii. **Precise Error Detection**:

SVAs capture specific protocol violations, such as missed acknowledgments or out-of-order transactions, reducing debugging time.

iii. **Reusable Assertions**:

Handshake protocol assertions can be easily reused across different designs, ensuring consistency in verification. [22,23].

4. 5 Optimized bus transaction mechanisms to reduce latency and improve APB's suitability for low-latency applications.

The Advanced Peripheral Bus (APB) is a simple and low-power bus designed for communication between SoC peripherals. However, its non-pipelined architecture and single-cycle transactions introduce higher latency, making it less suitable for real-time and high-speed applications.

4. 6 Enhancing Handshake Protocol Verification Through System Verilog Assertions

APB is low-power by design, there is significant room for optimization using dynamic power management strategies. These enhancements ensure better energy efficiency without affecting system performance

Chapter 5: Workdone

i. Generating the regression and coverage report.

ii. Finding the bugs and root cause in the design.

iii. Syncing with the corresponding configuration and feature owner.

iv. Running the failed test with the given fixes.

v. Generating UNR and exclusion file to improve the coverage.

vi. Setting up formal environment for required configuration.

vii. Ramping up with Fabric and Interface protocol .

viii. Data Transaction Mechanism between Agent and Fabric Interconnect.

ix. Design and Verification of APB Protocols.

Generating Reports:

Automate the Generation of Regression and Coverage Reports for Design Verification:

Automation is crucial to save time and improve efficiency in generating reports, especially for regression and coverage analysis. This involves creating scripts or tools that can automatically generate detailed reports on the outcomes of test runs, including whether design changes have affected the system's functionality.

Regression Reports:

Track the results of test executions over time, ensuring that new changes to the design don’t introduce bugs or regressions.

Coverage Reports: Provide insights into which parts of the design were exercised by tests. This helps identify areas of the design that were not tested and may require additional tests.

Bug Identification:

Identify Bugs and Analyse the Root Cause within the Design:
In this step, the primary focus is on finding and diagnosing bugs in the design under verification. You might use different verification techniques like simulation, formal verification, or manual inspection to detect errors.

The challenge is not just identifying the bug but understanding the root cause of the issue. For instance, is the bug due to a problem in the specification, the RTL code, or the verification environment? Root cause analysis ensures that the underlying issue is resolved, not just the symptom.

Collaboration with team:

Synchronize with Configuration Owner and Feature Owner for Debugging and Fixes:

Verification and debugging often require collaboration with various stakeholders. The configuration owner (who manages the design configurations) and the feature owner (responsible for specific design features or modules) play a crucial role in ensuring the design and verification processes align.

Regular communication is necessary for debugging, to clarify issues, review the causes, and implement fixes. This collaboration ensures that the changes made are aligned with the original design specifications and that the problem is properly addressed.

Test Re-running of failed tests:

Re-run the Failed Tests with the Applied Fixes to Ensure Issue Resolution:

After identifying and fixing a bug, it's essential to re-execute the tests that previously failed. This ensures that the fix has resolved the issue without causing any regressions.
Re-running tests also helps confirm that the fix has not inadvertently introduced new problems into the design. Run the necessary regression tests, formal checks, and property proofs to confirm the bug has been fixed without introducing new issues. Ensure functional and code coverage metrics are satisfied It ensures that the solution is effective across the full range of test cases.

Submit the changes for peer review. Colleagues or senior engineers review the fix to ensure it adheres to coding standards, is optimal, and maintains system stability.

Revalidate the fix in the relevant scenarios (unit tests, integration tests, or realworld workloads) to ensure its robustness. Document the bug details, the fix, and any test results for future reference. Update project logs, bug-tracking systems, or issue trackers.

Pull the latest version of the mainline to ensure your branch is up to date. Resolve any conflicts that arise during the merge or rebase process. Run a final regression to ensure your changes do not disrupt existing functionality.

Push the finalized, reviewed, and tested changes to the mainline repository. Use tools like Git for version control and ensure to include meaningful commit messages for clarity.

After successfully resolving the bugs, we push the change in mainline. Trigger a full regression on the mainline to confirm system-wide stability after integration. Address any issues promptly if they arise.

This process ensures the system remains stable, maintainable, and continues to meet performance and functionality expectations.

Coverage Analysis:

Create UNR (Unreachable Code Reports) and Exclusion Files to Optimize and Improve Coverage Results:

UNR Reports help identify parts of the design that are not being exercised during test execution, often due to design constraints or flaws. This provides an opportunity to add new tests or modify the design to ensure better test coverage.

UNR (Unreachable Code Reports) refers to a specific report generated during simulation, formal verification, or static analysis that identifies code within a design that is never executed or triggered under any condition. These reports are vital for optimizing hardware or software designs and ensuring their correctness and maintainability.

i. **Detection Methods**:

Static Analysis Tools: Identify unreachable lines of code by analysing the control flow without running simulations.

Coverage Tools: Highlight lines of code that were not executed during simulation.

Formal Verification: Finds UNR by proving certain conditions leading to the code cannot occur.

ii. **Sources of UNR in Hardware Design**:

- Incorrect coding styles or logic errors.
- Redundant or legacy code left over from previous revisions.
- Unused cases in case statements or FSM transitions.
- Overly constrained input conditions in formal properties.

iii. **Relevance to Verification**:

Coverage Metrics: UNR impacts coverage metrics like code coverage, functional coverage, and toggle coverage. It must be accounted for to assess the real progress of testing.

Quality Assurance: Unreachable code may indicate flaws in the design or the testbench.

Optimization: Identifying and removing UNR can lead to more efficient hardware and reduce synthesis and verification overhead.

iv. **Steps to Handle UNR:**

- **Generate Report:**
- Use tools like **VCS, Questa, JasperGold, or SpyGlass** to generate the UNR report.
- **Analyse Causes**:
- Check if the code is genuinely redundant.
- Determine if a testbench is incomplete and not stimulating certain conditions.
- **Classify UNR Code**:
- **True UNR**: Code that is genuinely unreachable due to design logic.
- **False UNR**: Code that is unreachable due to inadequate testbench scenarios.
- **Action Plan**:
- For **True UNR**, remove redundant code to simplify design.
- For **False UNR**, add test cases or update constraints in the testbench to improve stimulus.
- **Document**
- Maintain a log of UNR along with justification, whether it's fixed or waived. This is critical for audit and debugging.

Exclusion Files: These are used to mark parts of the design that should be intentionally excluded from coverage analysis. For example, some code may be unreachable due to known design constraints or certain conditions that are not meant to be tested.

Improving coverage helps provide confidence that the design is robust and has been thoroughly tested.

Understanding of SystemVerilog Assertions (SVAs):

Develop a Deep Understanding of SVAs for Implementing Simple Handshake Protocols:

SystemVerilog Assertions (SVAs) are powerful tools for formally verifying the correctness of a design. A deep understanding of SVAs is essential for writing assertions that ensure the design behaves as expected.

In particular, handshake protocols (communication mechanisms between modules) are common in digital design. Understanding how to write assertions to verify these protocols ensures that the data transfer between modules occurs correctly, without any errors or timing violations.

Writing SVA Code:

Write SVA Code Based on the Functional Specifications of the Model Under Verification:

After understanding the functional specifications of the design, SVAs are written to assert that the design meets these specifications.

Writing SVA code involves translating the functional requirements of the system (such as valid signal values, timing conditions, or sequences of operations) into assertions that can be checked automatically during simulation or formal verification.

Effective use of SVAs can help in catching design errors early in the verification process, improving the reliability of the design.

Example: A simple handshake protocol:

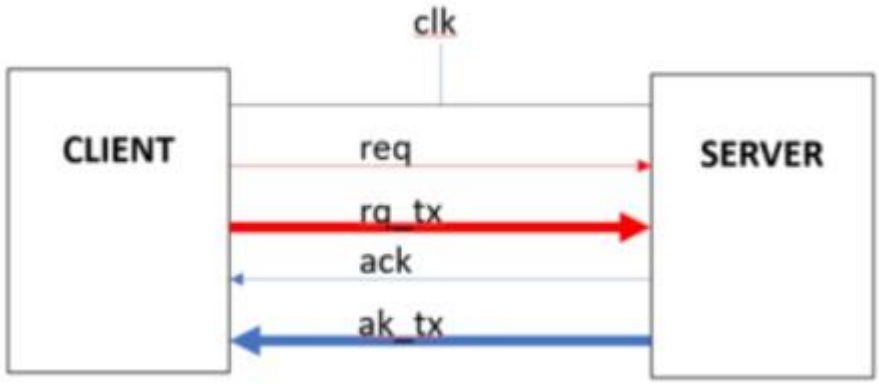


Figure 5.1 Handshake protocol

1. SINGLE PULSE REQ, SINGLE PULSE ACK

- The client and server are synchronized to a master clock.
- The request (req) and acknowledge (ack) signals are each single-bit signals.
- The client asserts a single pulse req signal when it needs access to a resource.

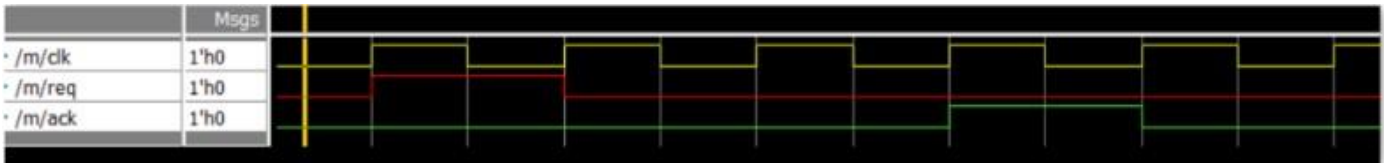


Figure 5.2 Handshake signal waveform

- The client shall wait for a single pulse ack signal. It expects the ack within 1 to 5 clock cycles, but
- not in the same cycle. Receipt of the ack completes the handshaking cycle.
- The client shall not assert another req signal while it is waiting for the ack signal as shown in figure 5.2.
- The server sets the ack signal for one pulse.
- The server shall not provide an ack signal without receiving a req signal.

SVA Code

i. The client and server are synchronized to a master clock.

ii. The request (req) and acknowledge (ack) signals are single-bit signals.

iii. The client asserts a single pulse req signal when it needs access to a resource

ap_req_pulse: assert property (@ (posedge clk) req |=>! req).

i. The server sets the ack signal for one pulse

ap_ack_pulse: assert property (@ (posedge clk) ack |=>! ack).

ii. The client shall wait for a single pulse ack signal. It expects the ack within 1 to 5 clock cycles, // but not in the same cycle. Receipt of the ack completes the handshaking cycles

ap_req_ack: assert property (@ (posedge clk) $rose(req) |->! ack ## [1:5] $rose(ack));

iii. The client shall not assert another req signal while it is waiting for the ack signal.

// a_no_initial_ack: Ensure that if a request (req) occurs, an acknowledgment (ack) must follow eventually. The assertion checks that if a request is present at any cycle, an acknowledgment should appear at some point in the future (after one or more cycles) without any initial acknowledgment being present at the start.

assert property (@ (posedge clk)! ack [*1:$] intersect req[->1]);

// ap_no_req_till_ack: Ensure that once a request (req) is asserted, no new requests are allowed until an acknowledgment (ack) is received. This assertion ensures that once a request is made, the system must not make another request until the current request has been acknowledged.

ap_no_req_till_ack: assert property (@ (posedge clk) req |=> !req[*1:$] intersect ack[->1]);

// **ap_no_ack_wo_req:** Ensure that an acknowledgment (ack) is never issued without a preceding request (req). This assertion checks that if an acknowledgment is present at any cycle, it must be preceded by at least one request (req) within the previous cycles.

ap_no_ack_wo_req: assert property (@ (posedge clk) not (ack ##1! Req [*0:$] ##1 ack));

SVA Verification:

Verify the Correctness of SVAs Using Tools Like **Jasper** for Formal Verification:

Once the SVA code is written, it must be verified to ensure that the assertions themselves are correct and do not contain logical errors. Tools like Jasper (a formal verification tool) can be used for this purpose.

Formal verification techniques check the design for exhaustiveness, meaning they prove that the assertions hold true across all possible states of the design. This is particularly useful for catching corner-case errors that might not be discovered through simulation alone.

Formal verification is a mathematical approach that rigorously checks the properties of the design, ensuring they hold in all situations, which is more comprehensive than traditional simulation-based verification.

By following this structured approach, you can ensure that the design is thoroughly verified, bugs are identified and resolved promptly, and the overall verification process is efficient and reliable.

- **Basic Overview of Agent Communication over Mesh Interconnect**

In modern System-on-Chip (SoC) architectures, efficient communication between a diverse set of processing agents is a fundamental design requirement. To address this, scalable and high-performance on-chip interconnects are employed to enable seamless data transfers between multiple system components. Among various interconnect topologies, the mesh interconnect has emerged as a popular choice due to its inherent scalability, flexibility, and bandwidth capabilities.

A high-level overview of agents communicating with one another through a mesh interconnect fabric. In this architecture, each agent connects to the mesh fabric via an intermediate bridge module. The agents represent various initiator or target components within the system, such as processor cores, memory controllers, input/output controllers, and specialized accelerators. These agents may operate at different clock frequencies and may adhere to different communication protocols, thereby requiring a common interface for system-wide connectivity.

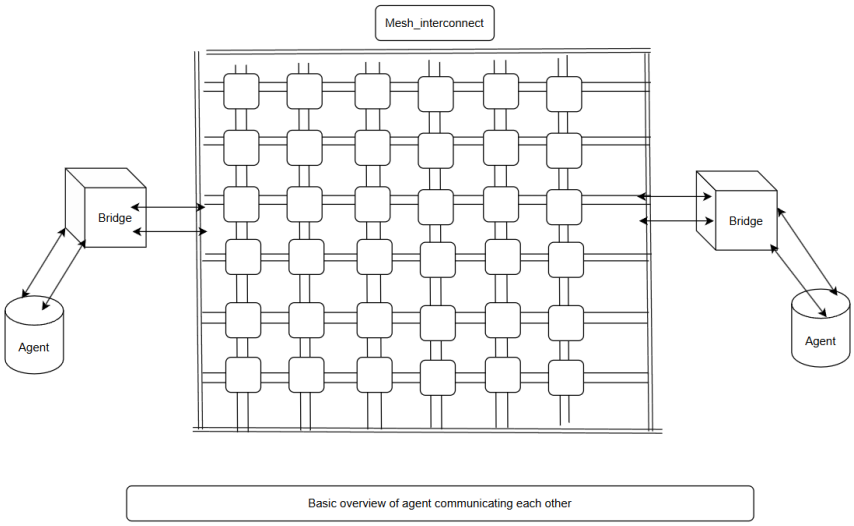


Figure 5.3 Basic overview of agents communication protocols

The bridge module as shown in figure 5.3 serves as this common interface and performs several critical functions to enable communication between the agents and the mesh fabric. It translates the native protocol of the agent into a fabric-compatible packet format and implements flow control mechanisms such as credit-based or token-based control to ensure reliable data transfer. The bridge also provides

buffering to manage latency mismatches between the agent and the mesh, and it packetizes data for optimal transmission through the fabric.

At the centre of this architecture is the mesh interconnect fabric itself. The fabric consists of a grid of interconnected routing nodes. These routers are capable of dynamically forwarding data packets toward their destinations based on the embedded addressing information. The mesh topology allows any two agents to communicate with each other through a path of intermediate routers, with multiple paths often available for load balancing and congestion management. The packet-switched nature of the mesh enables simultaneous communication between multiple pairs of agents, contributing to high aggregate bandwidth and improved system throughput.

Communication proceeds as follows. An agent initiates a transaction by sending data to its associated bridge module. The bridge translates and packetizes this data and injects it into the mesh fabric. The packet is then forwarded through one or more routing hops within the mesh until it reaches the destination bridge module. Upon receipt, the destination bridge depacketizes the transaction and translates it back into the appropriate format for the target agent, which then consumes the data.

The architecture illustrated in Figure 4.9 offers numerous advantages. The mesh topology supports seamless scalability; adding more agents simply involves extending the mesh by adding additional rows and columns of routing nodes. Bandwidth is naturally increased by the mesh's ability to support concurrent data flows along different paths, while latency is kept low through the use of short and direct routing paths. The use of bridge modules ensures that agents operating with different protocols and clock domains can participate in the mesh without requiring significant modification. Furthermore, the inherent redundancy of mesh routing paths enhances system robustness and fault tolerance, as traffic can be dynamically rerouted around failed links or congested areas.

In summary, the mesh-based interconnect architecture presented here provides an elegant and highly effective solution for facilitating high-performance communication between a large number of heterogeneous agents in a complex SoC. The use of programmable bridges ensures interoperability across diverse system components, while the mesh fabric delivers both high bandwidth and low latency communication. This architectural approach is well-suited for advanced multi-core processors, heterogeneous compute platforms, and server-class systems where performance scalability and communication efficiency are of paramount importance

- **Data Transaction Mechanism between Agent and Fabric Interconnect**

In modern multi-core systems and SoC architectures, efficient data transfer between processing agents and high-bandwidth interconnect fabrics is a critical requirement. To address this need, the architecture under study implements a credit-based flow control mechanism that enables reliable and scalable data communication between the agent subsystem and the fabric interconnect through a programmable bridge module.

The block diagram of this transaction mechanism is illustrated in Figure below. The architecture is composed of three primary components:

The Agents Block, where multiple initiator agents (such as CPU cores, I/O controllers, or accelerators) are located. The Bridge Module, which performs protocol translation, flow control management, and data buffering between heterogeneous interfaces. The Fabric Interconnect, a scalable interconnect mesh that routes data packets to their destination within the system.

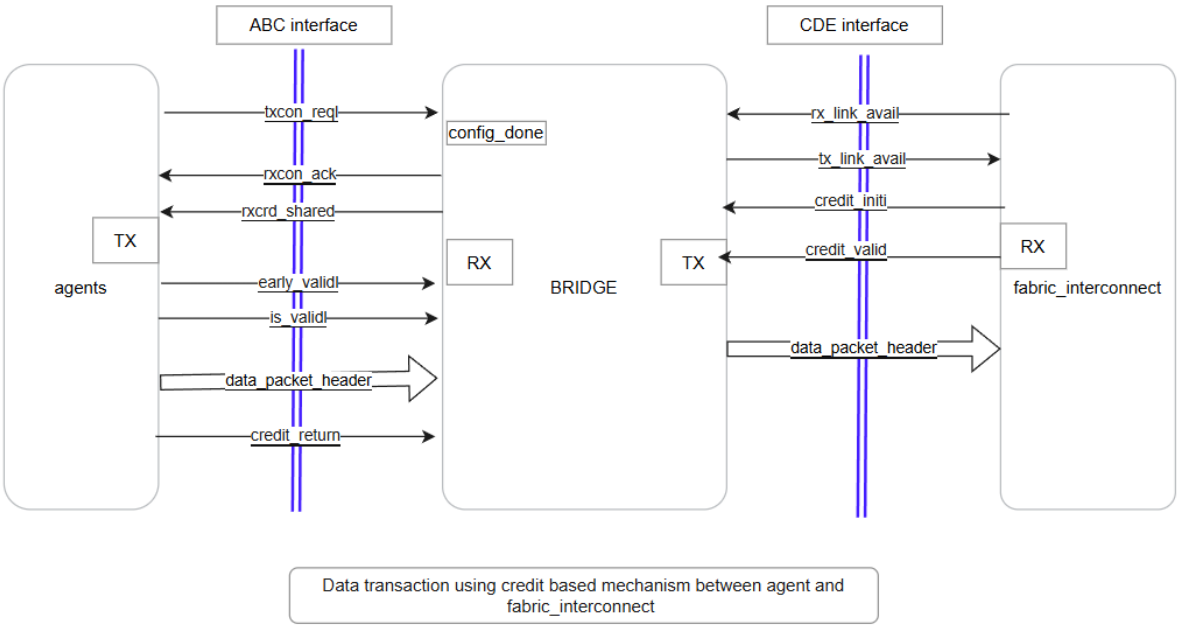


Figure 5.4.Transaction flow based on credit exchange mechanism

A.Agents Block (ABC Interface)

The Agents Block initiates data transactions towards the fabric. It interfaces with the bridge via the ABC interface, which carries both control and data signals. Each agent transmits requests through its internal TX (Transmit) module, as shown in the diagram.

The key signals as shown in figure 5.4 includes:

txcon_req: A connection request signal asserted by the agent to initiate a new transaction session.

rxcon_ack: An acknowledgment signal from the bridge indicating successful connection establishment.

rxcrd_shared: A shared credit signal that communicates available buffer capacity for managing flow control.

early_valid and is_valid: Signals indicating that valid data packets are ready for transmission.

data_packet_header: Encapsulated metadata associated with the transaction, including addressing, QoS, and routing information.

credit_return: A signal used by the bridge to return credits to the agent, thereby controlling the pacing of future transmissions.

config_done: A configuration status signal indicating that the bridge is ready to process incoming transactions.

B. Bridge Module

At the core of the architecture is the Bridge Module, which performs the following critical functions:

Protocol Translation: The bridge translates protocol semantics between the ABC interface (agent side) and the CDE interface (fabric side), ensuring compatibility across subsystems.

Flow Control Management: It implements credit-based flow control to prevent buffer overflows and ensure that data is only transmitted when the receiving side has sufficient capacity.

Buffering and Synchronization: The bridge temporarily buffers data to account for differences in clock domains, latency, and throughput between the agents and the fabric.

Data Path Management: It manages the RX (Receive) and TX (Transmit) paths internally to enable bidirectional flow of control and data signals.

The Bridge Module thus acts as an intelligent intermediary, ensuring smooth and efficient transaction flow.

C.Fabric Interconnect (CDE Interface)

On the other side of the bridge, the CDE interface connects the bridge to the Fabric Interconnect. The fabric serves as a high-bandwidth routing backbone that delivers packets to various endpoints within the system.

The signals involved in this interface include:

rx_link_avail: Indicates that the fabric's RX path is ready to accept incoming packets.

tx_link_avail: Indicates that the TX path from the bridge to the fabric is available for sending data.

credit_init: Used to initialize the credit flow control mechanism.

credit_valid: Signals the validity of current credit status.

data_packet_header: The encapsulated transaction header transmitted into the fabric interconnect.

D. Transaction Flow

The end-to-end transaction flow proceeds as follows:

Once active low reset asserted, an agent asserts **txcon_req** to request a transaction with the fabric.

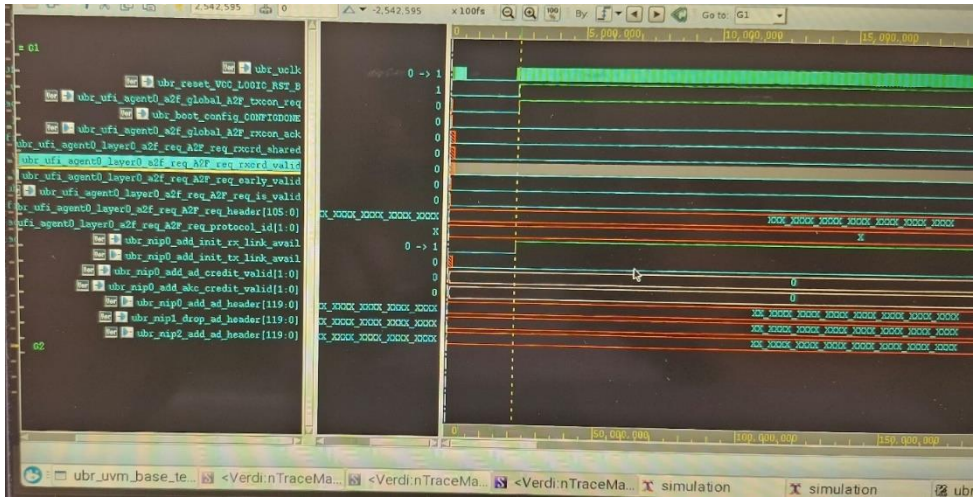


Figure 5.5.Agent sending connection request to bridge

Upon boot register configuration validation, the bridge responds with **rxcon_ack**, signaling readiness.

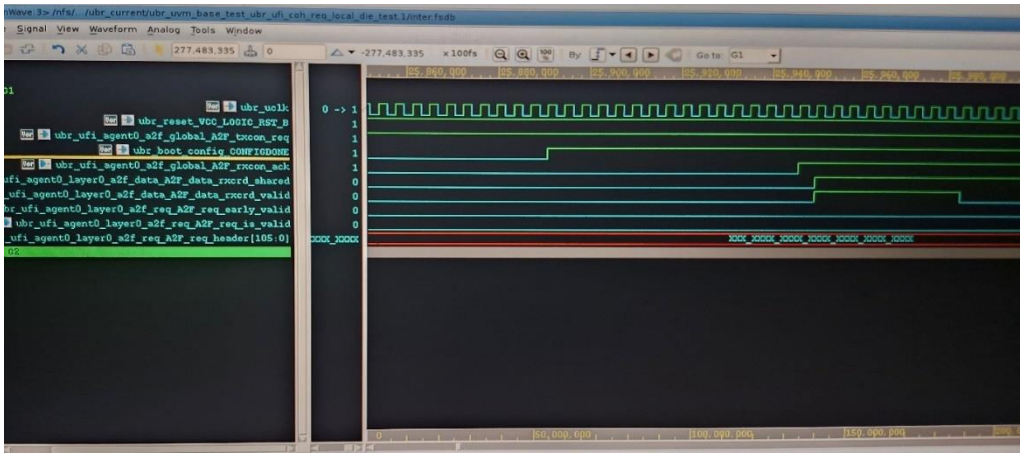


Figure 5.6. Bridge configuration validation and acknowledge the request from agent

Now credit initialization occurs shared credit and some dedicated credit. Flow control is established **via rxcrd_shared** and **credit_return** as shown in figure 5.5, 5.6 and 5.7 enabling the agent to transmit data at a pace compatible with fabric availability.

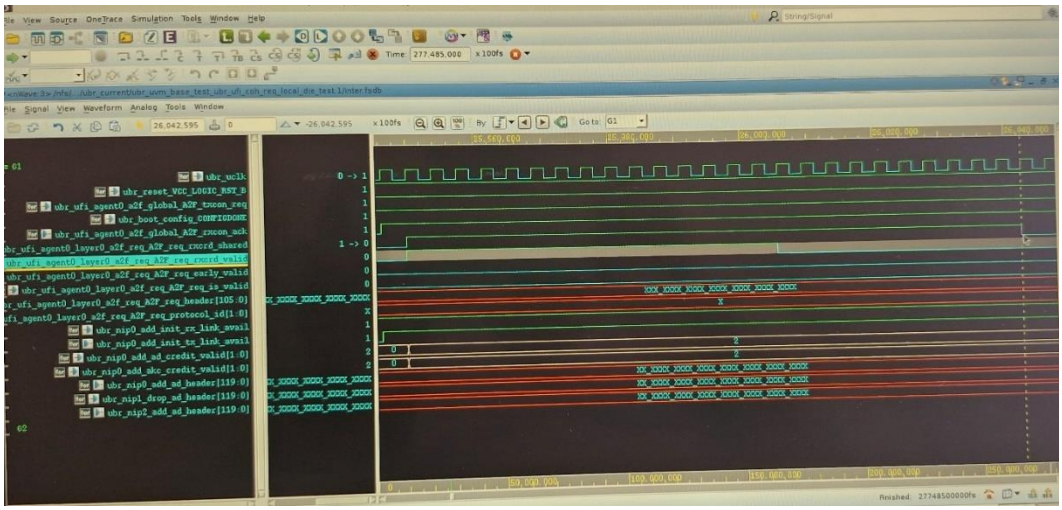


Figure 5.7. Credit advertised by bridge to agent

The agent sends valid data packets using **early_valid, is_valid**, and **data_packet_header** signals. The bridge processes and forwards the data towards the fabric interconnect via **data_packet_header(*--f207) sent** on the CDE interface.

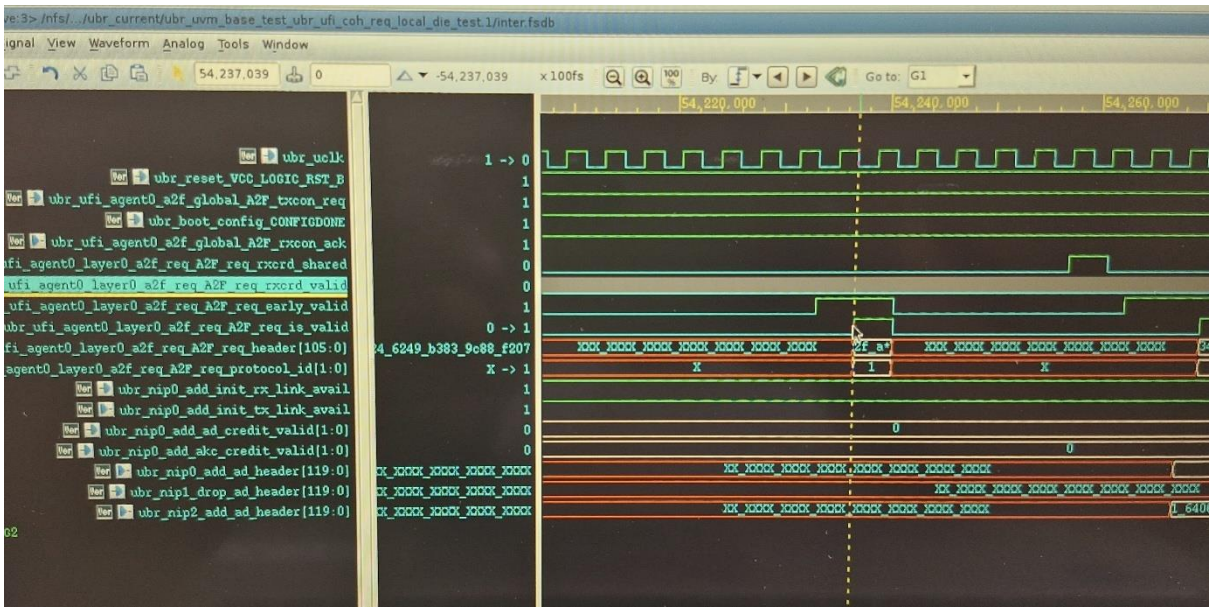


Figure 5.8. valid packet header transfer (f207)**

Fabric readiness and credit flow are continuously monitored through **rx_link_avail, tx_link_avail, credit_init**, and **credit_valid** signals as shown in figure 5.8 and figure 5.9.

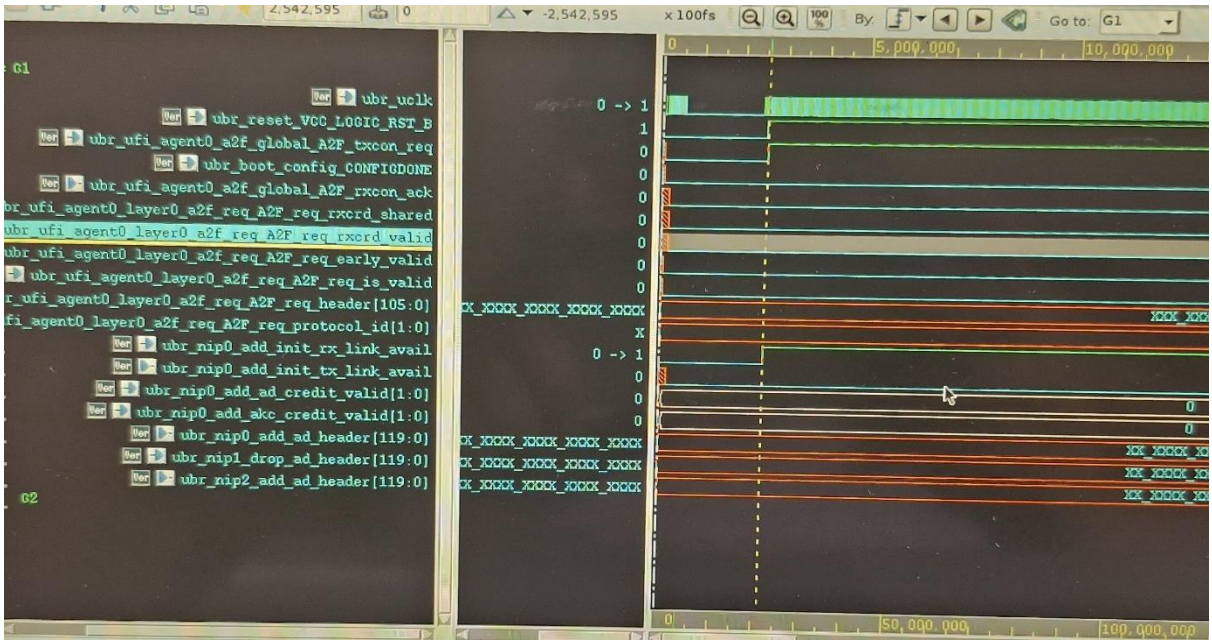


Figure 5.9 Link availability shared by mesh

At initial stage rx_lin_avail sent to TX and waiting for tx_link_avail.

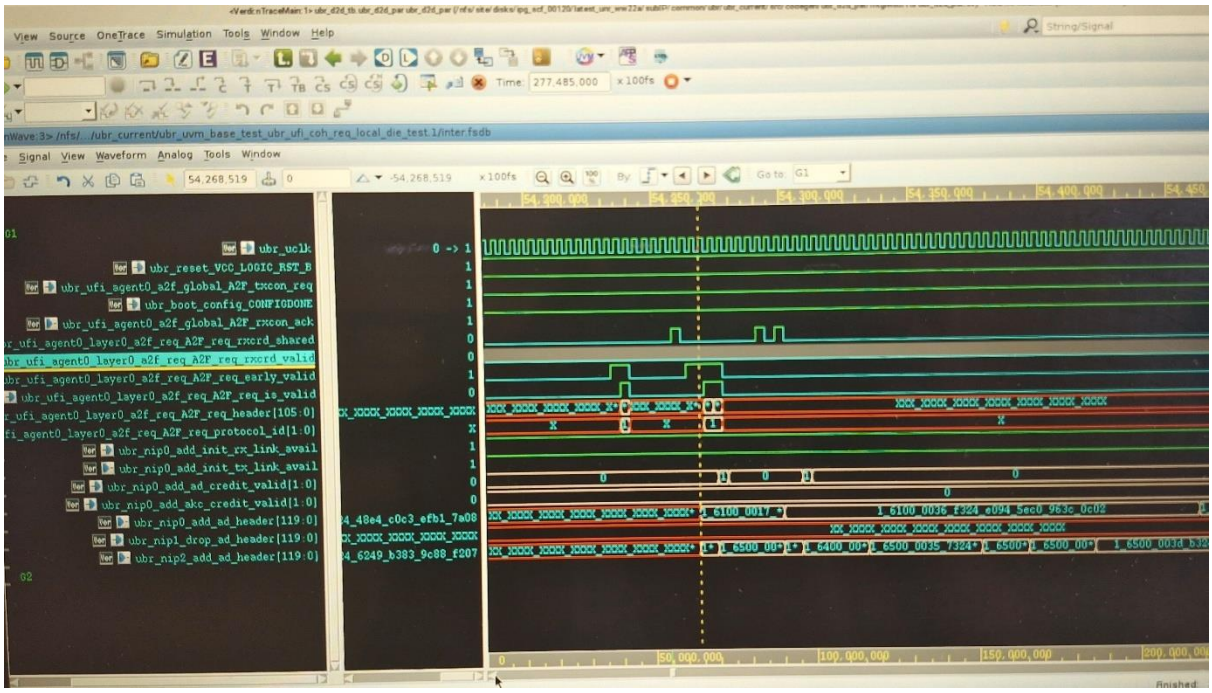


Figure 5.10. Header Packet(*_f207) received at mesh from agent

Once link established between the mesh and bridge as shown in figure 5.10, the packet sent from agent is received at the mesh interconnect and the according to the routing algorithm hops through and received by other agent via ABC, and CDE protocol. The transaction is completed when the data packets are successfully routed through the fabric to the target subsystem.

Key Debug Scenarios and Results

In early regressions, the bridge exhibited **incorrect credit handling**, which led to stalled transactions under sustained bidirectional traffic. In certain corner cases, **ordering violations** were observed for transactions targeting multiple agents with differing coherency requirements. The initial implementation failed to properly **drop unsupported transactions** under specific timing races between configuration updates and in-flight traffic. These issues were carefully debugged using waveform inspection, assertion failure traces, and transaction logs. Collaborative discussions with the design team led to refined bridge logic and improved corner-case handling. Subsequent regression runs demonstrated stable and correct bridge behavior under a wide range of operating conditions. Both **functional coverage** and **assertion coverage** reached closure, providing strong confidence in the bridge's readiness for integration into the full CPU platform

Design and Verification of APB protocol:

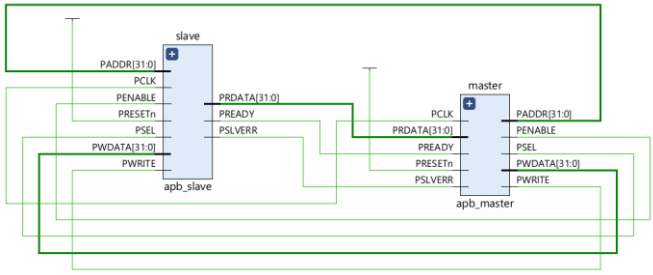


Figure 5.11 Design schematic of APB Protocol

APB SVA DESIGN CODE:

//APB Master

```
module apb_master (
    input wire PCLK, PRESETn,
    output reg [31:0] PADDR,
    output reg PWRITE, PSEL, PENABLE,
    output reg [31:0] PWDATA,
    input wire [31:0] PRDATA,
    input wire PREADY,
    input wire PSLVERR
);
    reg [1:0] state;
    parameter IDLE = 2'b00, SETUP = 2'b01, ACCESS = 2'b10;
    always @(posedge PCLK or negedge PRESETn) begin
        if (!PRESETn) begin
            state <= IDLE;
            PSEL <= 0;
            PENABLE <= 0;
            PWRITE <= 0;
            PADDR <= 0;
            PWDATA <= 0;
        end else begin
            case (state)
                IDLE: begin
                    PSEL <= 1;
                    PWRITE <= 1;  // Write operation
                    PADDR <= 32'h10;  // Address to write
                    PWDATA <= 32'hDEADBEEF;  // Write data
```

```
            state <= SETUP;
        end

        SETUP: begin
            PENABLE <= 1;
            state <= ACCESS;
        end
        ACCESS: begin
            if (PREADY) begin
                PENABLE <= 0;
                PSEL <= 0;
                state <= IDLE;
            end
        end
    endcase
  end
end
endmodule
```

// APB slave code

```
module apb_slave (
  input wire PCLK, PRESETn,
  input wire [31:0] PADDR,
  input wire PWRITE, PSEL, PENABLE,
  input wire [31:0] PWDATA,
  output reg [31:0] PRDATA,
  output reg PREADY,
  output reg PSLVERR
);
  reg [31:0] memory [0:255];  // Simulated memory for slave
  always @(posedge PCLK or negedge PRESETn) begin
    if (!PRESETn) begin
      PREADY <= 0;
      PSLVERR <= 0;
      PRDATA <= 0;
    end else begin
      if (PSEL && PENABLE) begin
        PREADY <= 1;  // Peripheral is read
```

```
            if (PWRITE) begin
                memory[PADDR] <= PWDATA;  // Write operation
                PSLVERR <= 0;
            end else begin
                PRDATA <= memory[PADDR];  // Read operation
            end
        end else begin
            PREADY <= 0;
        end
    end
end
```

Chapter 6 : Result

i. Daily Triaging and Report Generation:

A rigorous daily triaging process was conducted to identify and prioritize issues found during regression testing. Alongside, automated reports were generated for both regression results and coverage analysis, ensuring transparency and effective communication with the team. These reports provided critical insights into the test results and design behaviour, allowing the team to act promptly on any identified issues.

Generated an easy-to-read report summarizing results, highlighting priority issues, and providing actionable insights. Delivered reports in formats like Excel.

ii. Automation for report generation and running regression:

Objective: Streamline the process of analysing and reporting regression results.

Developed a script to parse regression logs and extract critical information (pass/fail status, test failures, coverage gaps, etc.). Automated root-cause analysis by correlating failure points with design features or recent changes. Integrated features to categorize failures (e.
g., setup issues, design bugs, testbench issues).

Designed a script to schedule and execute tests across various configurations automatically. Included features to monitor regression progress in real-time and handle resource allocation dynamically. Added retry mechanisms (recall) for tests that fail due to transient issues or infrastructure limitations.
Automated the collection and aggregation of test results from different simulation environments.

iii. Re-running the Failed Test Cases with Fixes and Updating the Mainline:

After identifying bugs or failures, fixes were applied, and the affected tests were re-executed. The re-run tests confirmed the resolution of the issues, and the mainline was updated accordingly, ensuring that the latest changes were integrated without introducing regressions. This iterative process helped maintain the stability of the verification flow and aligned the design with expected functionality.

iv. Generation of UNR Code and Coverage Analysis:

Unreachable code (UNR) reports were generated and analysed to identify coverage holes, enabling a more targeted approach to improving test coverage. Coverage gaps were filled by adding necessary tests and optimizing the verification environment. This approach ensured that all critical areas of the design were adequately tested, leading to better coverage analysis.

v. Regression and Coverage Improvement:

Coverage-driven verification effort, unreachable code (UNR) reports were generated and analyzed. By addressing coverage gaps through additional test development and refinement of the testbench environment, significant improvements in overall coverage were achieved. The project reached an **overall coverage** level of **approximately 98%,** as shown in table 6.2 indicating comprehensive exercise of the design and minimal uncovered areas. The **regression pass rate** also improved significantly, **reaching 99.5%,** as confirmed by the final regression reports in table 6.1. This demonstrated both the robustness of the verification process and the functional stability of the design.

Table 6.1 Regression Result

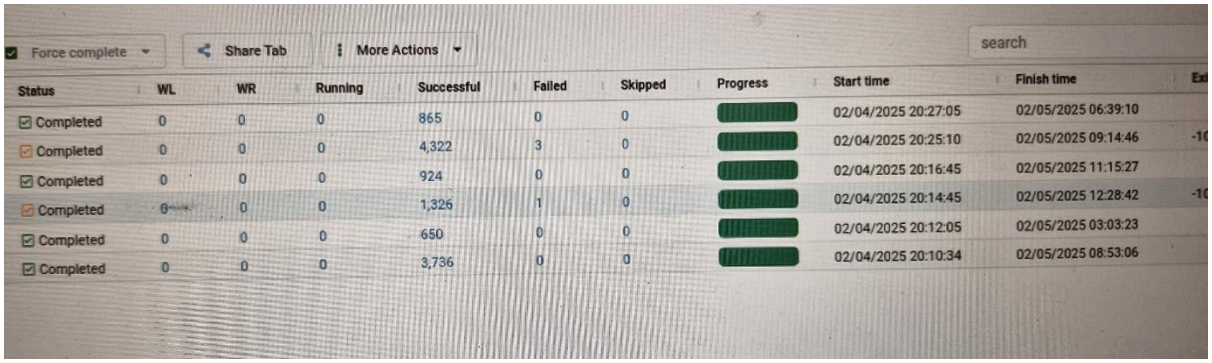


Table 6.2 Coverage

		4week_vdb					
		Shailendra			Hritik		
		manoj	siddartha	sukhjeet	arun		saikat
	Config	d2d	d2d_cbb	sca	uio1	uio1d	uio2
	line	98.38	98.5	98.92	98.57	99	98.77
	condition	92.33	91.01	95.72	92.85	91.17	93.38
	toggle	99.85	99.97	98.7	98.58	97.95	98.51
ww24.1	fsm	91.52	82.96	96.74	98.49	96.41	98.64
	branch	96.99	97.16	99.55	97.9	98.95	98.92
	assert	98.02	99.2	97.51	97.87	98.44	98.8
	group	97.92	98.5	98.98	94.12	98.61	74.07

<90% - Red
>90% and <95% - orange
>95% and <98% - yellow
>98% - Green

vi. Running Formal Regression and Analysing Failed Assertions:

Formal regression was also conducted to complement simulation-based testing. During this phase, formal tools verified key protocol properties and assertion checks, providing mathematical guarantees of correctness for critical design aspects. As shown in the formal regression summary, all **1188 assertions were successfully proven**, and **1386 coverage properties were achieved**, with no undetermined or failed checks. The formal results as shown in figure 6.3 reinforced the correctness of the design and provided confidence in its compliance with specified functional properties.

Formal assertion regression status:

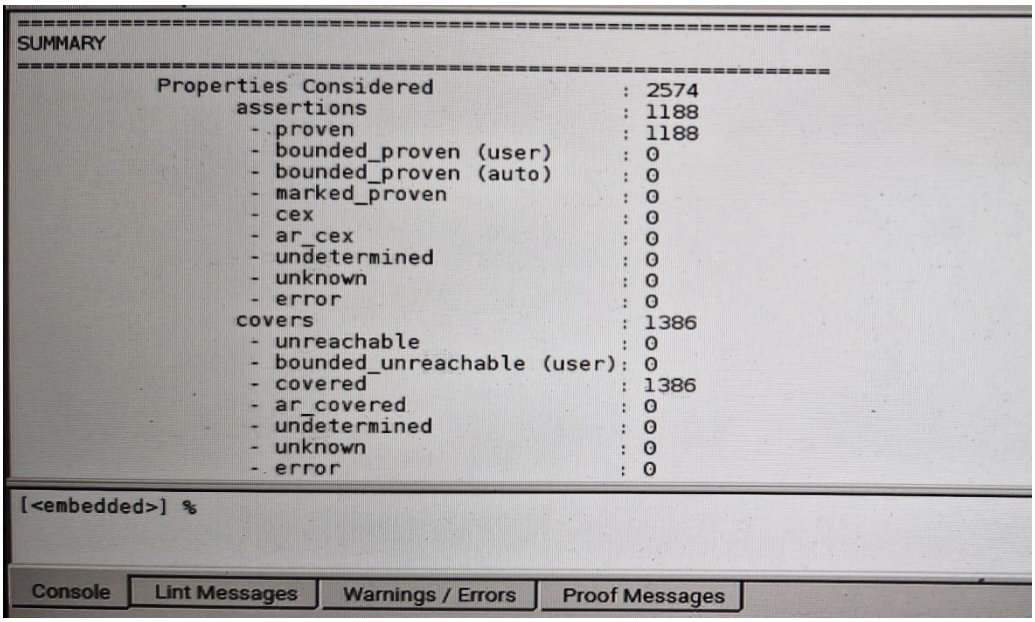


Figure 6.3 Formal assertion regression status

vii. Developing Formal Environment for Other Configurations:

A formal verification environment was developed for additional configurations, ensuring that all design variants were thoroughly validated. By extending the formal verification capabilities, the verification team was able to ensure consistency and correctness across multiple configurations of the design, thereby improving the overall robustness of the verification process.

viii. TCL console result as shown in figure 6.4 for APB read and write operation:

```
# run 1000ns
Time: 0 | PADDR: 00000000 | PWRITE: 0 | PSEL: 0 | PENABLE: 0 | PWDATA: 00000000 | PRDATA: 00000000 | PREADY: 0 | PSLVERR: 0
Time: 10000 | Writing to Address: 00000010, Data: dadadada
Time: 10000 | PADDR: 00000010 | PWRITE: 1 | PSEL: 1 | PENABLE: 0 | PWDATA: dadadada | PRDATA: 00000000 | PREADY: 0 | PSLVERR: 0
Time: 20000 | PADDR: 00000010 | PWRITE: 1 | PSEL: 1 | PENABLE: 1 | PWDATA: dadadada | PRDATA: 00000000 | PREADY: 0 | PSLVERR: 0
Time: 25000 | PADDR: 00000010 | PWRITE: 1 | PSEL: 1 | PENABLE: 1 | PWDATA: dadadada | PRDATA: 00000000 | PREADY: 1 | PSLVERR: 0
Time: 30000 | Write to Address: 00000010 Completed
Time: 30000 | Reading from Address: 00000010
Time: 30000 | PADDR: 00000010 | PWRITE: 0 | PSEL: 1 | PENABLE: 0 | PWDATA: dadadada | PRDATA: 00000000 | PREADY: 1 | PSLVERR: 0
Time: 35000 | PADDR: 00000010 | PWRITE: 0 | PSEL: 1 | PENABLE: 0 | PWDATA: dadadada | PRDATA: 00000000 | PREADY: 0 | PSLVERR: 0
Time: 40000 | PADDR: 00000010 | PWRITE: 0 | PSEL: 1 | PENABLE: 1 | PWDATA: dadadada | PRDATA: 00000000 | PREADY: 0 | PSLVERR: 0
Time: 45000 | PADDR: 00000010 | PWRITE: 0 | PSEL: 1 | PENABLE: 1 | PWDATA: deadbeef | PRDATA: dadadada | PREADY: 1 | PSLVERR: 0
Time: 50000 | Read from Address: 00000010 Completed, Data Received: dadadada
```

Figure 6.4 Simulated signal values of APB Protocol

ix. Simulated Waveform between APB master slave read and write operation shown in figure 6.5:

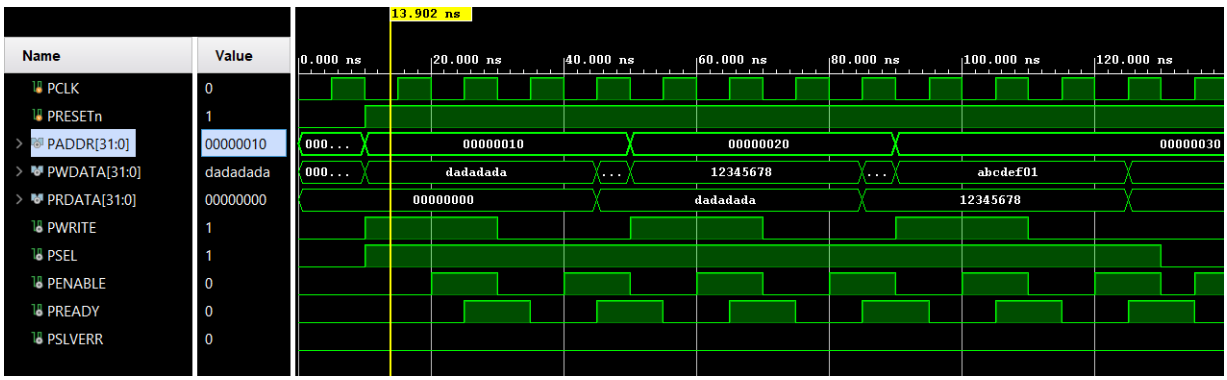


Figure 6.5 Simulated signal waveform of APB Protocol

x. Protocol validation of AMBA-APB protocol

Protocol validation was performed for the AMBA APB protocol within a UVM-based verification environment. A reusable testbench architecture was developed to verify the functional correctness and timing behavior of the APB slave design.

xi. Cache coherency protocols (MESI and MOESI)

Cache coherency protocols (MESI and MOESI) were validated within a simulated multi-core environment. The verification work demonstrated correct state transitions, ordering guarantees, and data consistency across multiple cores. Critical coherency scenarios were explored, and several subtle corner-case bugs were identified and resolved, resulting in a stable and fully coherent system.

Chapter 7: Conclusion

The verification process has evolved significantly, leveraging automation and strategic methodologies to ensure robust and reliable design outcomes. This comprehensive approach integrates daily triaging, automated report generation, targeted test re-execution, thorough coverage analysis, and advanced formal verification. Together, these elements form a powerful framework that efficiently identifies and resolves issues while enhancing overall design quality.

A key milestone is achieving 98% coverage, validating nearly all functional scenarios and corner cases. This accomplishment highlights the team's dedication to systematically addressing test coverage gaps. Automation for generating triaging reports has streamlined debugging, enabling quick failure analysis and prioritization. This reduction in manual intervention allows engineers to focus on problem-solving and innovation.

Additionally, the regression pass rate has reached an impressive 99.5%, reflecting the rigor of the automated testing framework. Automated regression suites, tailored for diverse configurations, coupled with efficient mechanisms for rerunning failed tests, have minimized delays and enhanced verification cycle efficiency. Notifications and detailed reports keep stakeholders informed, fostering collaboration and quick decision-making.

Formal verification has been instrumental in addressing failed assertions and proving design properties. Tailored formal environments have enabled comprehensive validation across design variants, uncovering potential issues early and ensuring correctness across use cases.

These achievements underscore a robust verification framework that systematically improves coverage and validates functionality with minimal risk of undetected bugs. The process ensures design reliability, meeting stringent performance standards. By setting a benchmark of 98% coverage and a 99.5% regression pass rate, the team has demonstrated its commitment to excellence.

This approach not only ensures a reliable current design but also establishes a strong foundation for future verification, emphasizing continuous improvement and innovation.

After implementing Design and Verification AMBA APB Protocol through Verilog, implement the same through UVM. It will give deep insight of verification methodology used in IP verification and ultimately get the good grip on verification languages.

References

[1] "Importance of Early Design Verification in SoC Development. " Journal of Semiconductor Technology, 2021.

[2] "Lee, C. "Cost Implications of Late-Stage Design Bugs. " IEEE Transactions on VLSI Systems, 2019.

[3] "Brown, A., et al. "Trends in Design Verification: Challenges and Best Practices. " Verification Journal, 2022.

[4] "Formal Verification in Modern IP Design. " Design Automation Conference Proceedings, 2020.

[5] Wang, H. , et al. "Machine Learning-Driven Coverage Closure in SystemVerilog Environments. " International Symposium on VLSI Design, 2020.

[6] Gupta, A. , et al. "Reinforcement Learning for Dynamic Covergroup Adjustment in Functional Verification. " Journal of Systems and Software, 2021.

[7] Iyer, S. , et al. "Adaptive Covergroup Generation Using Simulation Profiling. " Design, Automation & Test in Europe Conference (DATE), 2021

[8] Gupta, R. , et al. "Parameterized Coverage for Configurable IPs: Dynamic Adjustments. " IEEE Embedded Systems Letters, 2022.

[9] G.Kumar, S.Verma and R. Sharma, "UVM-based verification methodology for AMBA APB in SoC environments," in Proceedings of the IEEE International Conference on VLSI Design and Embedded Systems (VLSI-ES), 2017, pp. 45-50.

[10] P. Mishra, A. Gupta, and T. Reddy, "Comparative analysis of AMBA APB, AXI, and AHB protocols for latency, throughput, and power efficiency," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 22, no. 8, pp. 1550-1560, Aug. 2014.

[11] Accellera Systems Initiative, *UVM Class Reference Manual*, Accellera, 2020

[12] "The Universal Verification Methodology (UVM) Framework: Industry Adoption and Evolution. " IEEE Design & Test of Computers, 2021.

[13] IEEE Standard for System Verilog: IEEE Std 1800-2012, Defining the coverage model: covergroup" IEEE Computer Society and IEEE Standards Association Corporate Advisory Group. Web. 21 Feb. 2013.

[14] M. Mostafa, M. Safar, M. W. El-Kharashi and M. Dessouky, "System Verilog Assertion Debugging Based on Visualization, Simulation Results, and Mutation," 2014 15th International Microprocessor Test and Verification Workshop, Austin, TX, USA, 2014.

[15] P. Gurha and R. R. Khandelwal, "SystemVerilog Assertion Based Verification of A MBA-AHB," 2016 International Conference on Micro Electronics and Telecommunication Engineering (ICMETE), Ghaziabad,India, 2016

[16] A. Banerjee, "Effective SystemVerilog Functional Coverage: Design and Coding Recommendations," Proceedings of the DVCon India, Bengaluru, India, 2019.

[17] N. Gupta and P. Mishra, "Practical Hacks for SystemVerilog Coverage," Tessolve Technical Reports, 2021.

[18] M. McKinney, "Python for Data Analysis," 2nd ed., O'Reilly Media, 2017.

[19] W. McKinney, "Data Wrangling with Pandas, NumPy, and Python," O'Reilly, 2018.

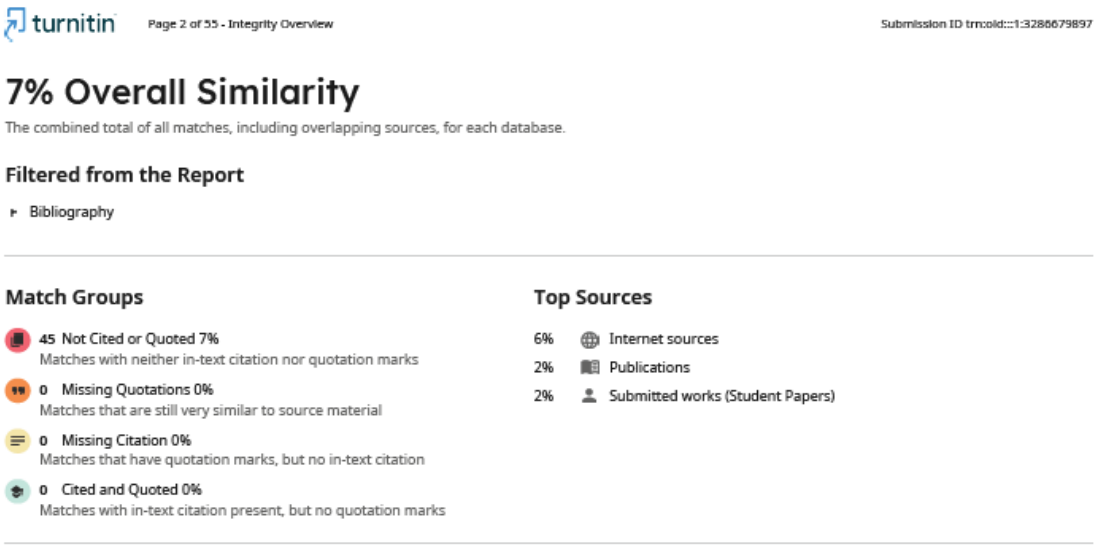
[20] M. Purandare, "Waveform Debugging Techniques in Chip Design," Journal of VLSI Design Verification, vol. 8, no. 3, pp. 45-56, 2020.

[21] T. Mitra, "Efficient Testbench Debugging Using Assertion-Based Debugging," IEEE Design & Test of Computers, vol. 36, no. 2, pp. 23-31, 2019.

[22] S. A. Edwards, "SystemVerilog Assertions for Protocol Compliance," Proceedings of the IEEE International Conference on Verification and Validation, 2018, pp. 101-109.

[23] B. Chandran, "Reusable Assertions for Communication Protocols in UVM," IEEE Transactions on CAD of Integrated Circuits and Systems, vol. 40, no. 2, pp. 345-352, 2021.

Plagiarism report



turnitin Page 2 of 55 - Integrity Overview Submission ID trn:oid:::1:3288679897

7% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

Filtered from the Report

- Bibliography

Match Groups

- 45 Not Cited or Quoted 7%
  Matches with neither in-text citation nor quotation marks
- 0 Missing Quotations 0%
  Matches that are still very similar to source material
- 0 Missing Citation 0%
  Matches that have quotation marks, but no in-text citation
- 0 Cited and Quoted 0%
  Matches with in-text citation present, but no quotation marks

Top Sources

6% Internet sources
2% Publications
2% Submitted works (Student Papers)