Trends in Register Verification with a Focus on IP Employed in Networking Devices

A project report submitted in partial fulfilment of the requirement for the Award of the Degree of

MASTER OF TECHNOLOGY

In VLSI DESIGN

Submitted By

ADITYA BAHUGUNA

602362043

Under the Supervision of

Dr. VINAY KUMAR & Dr. POONAM VERMA

Professor & Assistant Professor, ECED, TIET

Mr. PRANAV B. JOSHI

Engineer Manager, IP DV, Intel

THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be University)

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

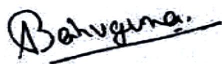
THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY

(A DEEMED TO BE UNIVERSITY), PATIALA, PUNJAB

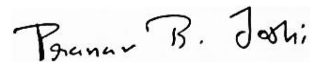
JULY 2025

DECLARATION

I, **Aditya Bahuguna** hereby declare that the work presented in this report entitled **"Trends and Predictions in Register Verification with a Focus on IP Employed in Networking Device"** in partial fulfilment of the requirement for the award of degree of **Master of Technology (VLSI Design**) submitted at **Electronics and Communication Engineering Department**, Thapar Institute of Engineering & Technology (Deemed to be University), Patiala is an authentic record of work carried out under supervision of Industry Mentor **Mr. Pranav B. Joshi (**IP DV Manager, Intel Technology India Private Limited**), Dr. Vinay Kumar** (Professor, Electronics and Communication Engineering Department, Thapar Institute of Engineering & Technology) and **Dr. Poonam Verma (**Assistant Professor, Electronics and Communication Engineering Department, Thapar Institute of Engineering & Technology) from **June 2024** to **June 2025**. The matter presented in this has not been submitted in part or in whole to any other university or institute for the award of any other degree, as per my knowledge.



Aditya Bahuguna
602362043
Date: 17-06-2025



Mr. Pranav B. Joshi
IP Design Verification Manager,
DV Architect, Intel Technology
India Private Limited, Bengaluru
Date: 17-06-2025

Dr. Vinay Kumar
Professor,
Department of Electronics and Communication Engineering, Thapar Institute of Engineering & Technology, Patiala
Date: 02-07-2025

Dr. Poonam Verma
Assistant Professor,
Department of Electronics and Communication Engineering, Thapar Institute of Engineering & Technology, Patiala
Date: 02-07-2025

ACKNOWLEDGEMENT

This report wouldn't have come together without the support and guidance of several people who helped me along the way.

I'm really thankful to Dr. Vinay Kumar, Professor, ECED for being there with advice, and technical insight. His support made a big difference both the research and writing stages.

I would also like to sincerely thank Dr. Poonam Verma, Assistant Professor, ECED my co-supervisor, for her thoughtful feedback. Her advice enabled me to maintain focus and improve my strategy at crucial points.

I'm thankful to Mr. Pranav B. Joshi, Manager of the NCNG team at SIE FNIC/NESG IP DV, Intel Pvt Ltd, for the opportunity to intern with his team. His guidance really shaped my learning experience, and I appreciated how thoughtful and supportive he was about my academic goals.

A special thanks to Mr. Aadhitya S V, IP Verification Engineer at Intel, for his patience and steady support throughout the project. His feedback helped me stay focused, and our technical discussions really helped me understand the work more clearly. His insightful critiques throughout various phases of my work were intellectually stimulating and assisted me in honing my ideas. I am deeply grateful to him for all the discussions that helped me understand the technical details of my work.

Last but not least, my family and dear friends. None of this would have been possible without their love. And the one above all of us, the omnipresent God, for giving me the strength to plod on, thank you so much, everyone.

ABSTRACT

As digital systems continue to grow in complexity, especially within networking and SoC domains, validating hardware before fabrication has become a pressing necessity. This project focused on verifying the register interface of a Data Aligner IP, a module frequently used in packet-based communication systems. The IP had a set of programmable registers used for both data path alignment and control. Since the register behavior had to remain consistent even when traffic patterns or timing conditions changed, validating this wasn't trivial. Early in the project, I noticed that manual register access and checking was becoming repetitive and hard to scale. So I put together a UVM-based testbench with SystemVerilog and integrated a RAL model to simplify the entire process.

The Register Abstraction Layer (RAL) was integrated after experimenting with a few manual approaches, which proved time-consuming and error-prone. By transitioning to a structured, script-based generation flow (using spreadsheet input for spec definition), I was able to create an accurate register model, integrate it into the environment, and perform functional coverage analysis efficiently.

Assertions were embedded early on to catch design mismatches, and constrained-random sequences were written to stress corner cases. We simulated the register interface, which made debug much faster. Instead of handling register behavior manually—something that quickly became tedious and error-prone—I used RAL to automate how access sequences were built and how checks were run.

During development, the register specification kept changing—sometimes just small updates, sometimes entire fields added or removed. Instead of constantly modifying sequences or rewriting checkers every time that happened, I worked on making the testbench flexible enough to handle these shifts. The point wasn't to nail everything on day one—I just didn't want to waste time redoing the same stuff every time the spec changed. So I set things up with enough flexibility that small changes—like renaming a field or adjusting a default—wouldn't break the whole environment. It wasn't some perfect setup, but it worked, and over time it definitely reduced the amount of rework and made life a bit easier as the project moved forward.

TABLE OF CONTENTS

Sr. No	Name of the Chapters	Page No
	Pre-pages………………………………………………………………………….	
	Declaration……………………………………………………………………….	*ii*
	Acknowledgement…………………………………………………………………	*iii*
	Abstract………………………………………………………………………..…	*iv*
	List of Tables…………………………………………………………….……….	*viii*
	List of Figures……………………………………………………………………	*ix*
Chapter 1	Introduction	10
1.1	Overview…………………………………………………………	10
1.2	Motivation………………………………………………………..	11
1.3	Verification Fundamentals……..……………………………………….	12
1.4	Verification Methodologies…..……………………………………….	12
1.4.1	Simulation-Based Verification..……………………………………….	13
1.4.2	Formal Verification…………………………………………………….	13
1.5	Verification Challenges……...………………………………………..	13
Chapter 2	Literature Review	15
2.1	Register Verification………………………………………………….	15
2.2	Importance of Register Verification…………………………………..	15
2.3	Traditional Workflow………………………………………………….	16
2.3.1	Creating a Test Plan……...………………………………………….	17
2.3.2	Verification Methodology……..……………………………………….	17
2.3.3	Efficient Test Case Design..…………………………………………….	17
2.3.4	Addressing Edge Cases ……...………………………………………..	17
2.4	Literature Survey………………………………………………………	17
2.5	Problem Formulation………………………………………………….	21
Chapter 3	Register Model and Integration Overview……………………………..	23
3.1	Overview………...………………………………………………....	23

3.2 Register Specification………………………………………………… 22

3.3 Register Model Use Flow…………………………………..……….. 24

3.4 Register Model Integration Overview…………………………………. 25

Chapter 4 Implementation of Register Abstraction Layer (RAL) 26

4.1 Adapter…………………………………………………………………. 27

4.2 Bus Agent………………………………………………………………. 27

4.3 Predictor……………………………………...………………. 27

4.4 Front-door and Back-door access……………………………..…… 27

4.5 Test and Test Sequence……………………………………….. 28

4.6 Handling Special Cases………………………………………….. 28

4.6.1 Lock Bits……………………………………………………… 28

4.6.2 Write-Only Registers………………………………………………… 28

4.6.3 Special Bits…………………………………………………………… 29

4.7 Implementation Algorithms………………………………………….. 29

4.7.1 Test Algorithm…………………….…………………………………… 29

4.7.2 Test Sequence Algorithm……………………………………………… 29

4.7.3 RAL Access Sequence Algorithm……………………………………... 30

Chapter 5 Work Done and Result Discussion 31

5.1 Test Plan…………………………………………………………. 31

5.2 Register Testing Summary…………………………………………… 32

5.2.1 Reset Value Verification………………………………………… 32

5.2.2 Attribute Register Access – Mode-Based Testing………………….. 32

5.2.3 Register Access Order Independence…………………………………… 32

5.2.4 Address Space Access Validation……………………………………….. 32

5.2.5 Interface-Level Access and Error Handling…………………………… 32

5.2.6 Back-to-Back Read Operation Verification…………………………….. 32

5.2.7 Back-to-Back Write Operation Verification …………………………… 32

5.3 Tools Used in Verification…………………………………...……….. 33

5.3.1 Synopsys VCS……………………………………………………… 33

5.3.2 Synopsys Verdi………………………………………………………….. 33

5.4 Result and Wave…………………..…………………..………….. 33

5.5 Design Bugs and Solution…………………………………………….. 36

Chapter 5 Conclusion and Future Scope 39

5.1 Conclusion………………………………………………………… 38

5.2 Future Scope……………………………………………………… 38

References……………………………………………………………..…………… 40

LIST OF TABLES

Sr. No	Table Details	Page No
Table 3.1	*Attributes Overview……….…………………………………………*	*24*
Table 3.2	*Integration Components………………………..………………......*	*25*
Table 5.1	*Test-plan……………………………………………………………*	*31*
Table 5.2	*Different modes for the Attribute test.………………………………*	*31*
Table 5.2	*Summary of Identified Design Bugs and Implemented Solutions*	*37*

LISTS OF FIGURES

Sr. No	Table Details	Page No
Figure 1.1	*Design flow*…………………………………………………………..	*11*
Figure 5.4	*Traditional Flow*……………………………………………….	*16*
Figure 4.1	*Register Abstraction Layer*…………………………………..	*26*
Figure 5.1	*Reset Test*………………………………………………………	*34*
Figure 5.2	*Attribute Test Mode_0*……………………………………….	*34*
Figure 5.3	*Attribute Test Mode_1*………………………………………………..	*34*
Figure 5.4	*Attribute Test Mode_2*……………………………………………	*34*
Figure 5.5	*Attribute Test Mode_3*……………………………………………	*34*
Figure 5.6	*Varying Sequence Test*…………………………………………..	*35*
Figure 5.7	*Address spaces Test*……………………………………………	*35*
Figure 5.8	*Accessible interface Test*……………………………………………..	*36*
Figure 5.9	*B2B Read Test*…………………………………………………….	*36*
Figure 5.10	*B2B Write Test*……………………………………………………………….	*36*
Figure 5.11	*Error Description*………………………………………………..	*38*

Chapter 1

INTRODUCTION

This chapter talks about the main ideas underlying hardware design verification and why it's an important part of making sure that digital systems work as they should. It talks about why verification is important, the most frequent methods utilized, and the problems that come up when making current ASICs and SoCs. The point of this is to get the report started, which is about how to make verification faster by using automation.

1.1 Overview of Design Verification

Verification is an important step in the development of ASIC (Application-Specific Integrated Circuit) and SoC (System-on-Chip) since it makes sure that a design works as planned before it goes into production. As chips get more complicated, the need for strong verification methodologies grows as well. Question's such as "Is this feature fully tested?", "How will new functionality be validated?" and "What defects exist and how were they detected?" are central to the development lifecycle.

Verification isn't something you do just once. It starts early, frequently during the architectural or microarchitectural planning phase, and goes on throughout the design process. The major purpose is to make sure that the design fits all of the important needs, such as safety, functionality, performance, and energy efficiency. Formal verification, power-aware simulations, static analysis, and FPGA prototyping are now used along with traditional RTL (Register Transfer Level) simulation to make sure that all areas are covered.

If you don't find design faults before tape-out, you could have to do expensive re-spins, launch products later, and lose a lot of money. So, in chip design, verification is not only a technical need, but also a strategic one.

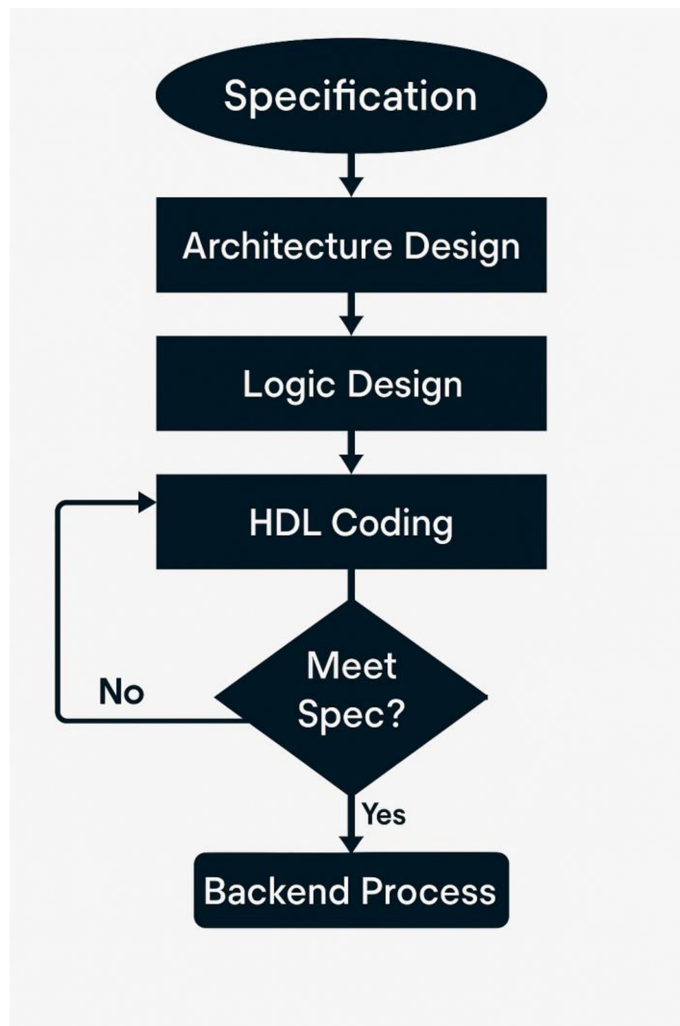


Figure 1.1: Design Flow

1.2 Motivation

Functional verification has become one of the most important—and time-consuming—parts of the design process as digital systems become more complex. With devices packing in more features and pushing for higher performance, it's essential to use effective verification strategies to ensure everything works as expected.

The main goal of this study is to improve the verification process, especially for reusable IP blocks that are essential for managing data flow in networking and communication systems. The focus is on finding smarter, more automated methods that reduce manual effort, increase coverage, and speed up verification.

A major part of this project involves building a Register Abstraction Layer using a custom SystemVerilog class library. This approach is based on the UVM (Universal Verification Methodology) register model and is aimed at making register-level verification faster, more adaptable, and easier to manage.

1.3 Verification Fundamentals

Verification is basically the opposite of design. It starts with an implementation and works backwards to make sure it meets the original requirements. Mistakes can happen because the implementation is wrong or because the specifications are wrong. These problems can happen because people misunderstand things, the requirements aren't clear, or the documentation isn't complete.

Redundancy is often used to lower these kinds of risks. For example, using different methods to carry out the same specification and then comparing the results can help find inconsistencies. But this method is limited by cost and complexity, so it is only used a few times.

Another way to do this is to use synthesis tools that make implementations straight from specifications. This method looks good, but it has some problems because high-level specifications are often too vague and don't include the timing and structural details needed for accurate synthesis.

1.4 Verification Methodologies

A strong verification method starts with a clear test plan that lists the features, operations, and edge cases that need to be checked. Using scoreboarding methods, we keep track of progress, and we use coverage metrics, mainly functional and code coverage, to measure the quality of verification.

There are three main ways to do verification:

1.4.1 Simulation-Based Verification

This is the method that most people use. It means putting the RTL design into a testbench, sending it input signals, and then checking the output against what you expected it to be. You can make stimuli and reference outputs ahead of time or while the simulation is going on. There are two types of simulation tools: event-driven and cycle-based. Each has its own performance characteristics that depend on how complicated the design is and how the clock domain is set up.

1.4.2 Formal Verification

Unlike simulation, formal methods don't use test vectors. They don't do this; instead, they use math to prove or disprove things about the design:

- **Equivalence Checking**: This checks to see if two versions of a design, such as RTL and gate-level netlist, do the same thing.

- **Property Checking**: This checks that the design meets certain requirements using tools like model checkers.

Formal methods are powerful, but they have problems like figuring out what the right constraints are and fixing properties that are written in a language other than the design.

1.5 Verification Challenges

Even with better tools and more efficient workflows, it's still hard to keep track of how chips work. Here are a few important things to remember:

- **Scenario Coverage**: It's hard to think of every possible way a system could act, and even harder to set up all those situations in a test environment. If the verification environment isn't rigorous and flexible, a lot of edge cases or strange situations can sneak through.

- **Testbench Complexity**: Making and keeping up with random testbenches is a lot of work.

- **Coverage Closure**: Getting the last 20% of coverage often takes a lot more work than the first 80%.

- **Manual Iterations**: It can take a lot of time and effort to debug and improve tests.

Because of these problems low-tech verification methods and automation are needed. Industry trends are moving more and more toward making verification faster and cheaper by using reusable parts, smart test generation, and automated environments.

Chapter 2

LITERATURE REVIEW

In this chapter, the basics of Register verification are discussed, and the literature of the existing research works is presented.

2.1 Register Verification

Even the smallest components in a design, like configuration registers, can have a major impact on system functionality. These registers are critical for ensuring that the design behaves as intended, and verifying each bit and field is crucial, especially in complex SoCs or IPs where registers and memory blocks make up a significant portion of the architecture.

Registers serve as the interface between software and hardware, and their accuracy is vital. As the number of registers grows, so does the difficulty of keeping documentation, implementation, and maintenance aligned. Manual handling increases the risk of errors and reduces productivity.

Luckily, registers usually have a structure that is easy to guess based on the attributes of the fields. This makes it possible to use a high-level description framework like RAL to set up register architecture. This makes it possible to automatically create design code, documentation, and verification components. This speeds up the process and makes sure that all design, verification, and firmware teams are on the same page.

2.2 Importance of Register Verification

For example, a 32-bit Base Address Register (BAR) is used to map memory in host interface protocols. You can tell how much memory the device needs by looking at the number of read-only bits in the BAR. When the host sends 0xFFFFFFFF and gets back 0xFFC00000, it means that 2 MB of memory is needed. If there is a design flaw that allows a read-only bit to be written to, though, the readback could give back 0xFFE00000, which is only 1 MB. This difference can cut the amount of RAM in half, which can lead to problems that are hard to find later in the development process.

2.3 Traditional Workflow

The usual way to do things is for the architecture team to write up specifications first. When these are done, hardware, software, and verification engineers each put their own ideas about the register map into action. Tests are made and run as the design moves forward. But when bugs are found or features change—whether because of design flaws or marketing requests—everyone on all teams needs to make updates. These updates are often not done at the same time, which causes the design, documentation, and test environments to be out of sync.

In big designs, making RTL for hundreds of registers with different field attributes can take weeks. After that, verification engineers spend more time making environments that can be used again and writing tests for reset values, checking read/write access, and checking attributes. If you change the address maps or register definitions, you have to make more changes, which takes a lot of time and is easy to mess up.

This broken flow can cause problems and slow things down. Automating the process of designing and checking the register can help with these problems.

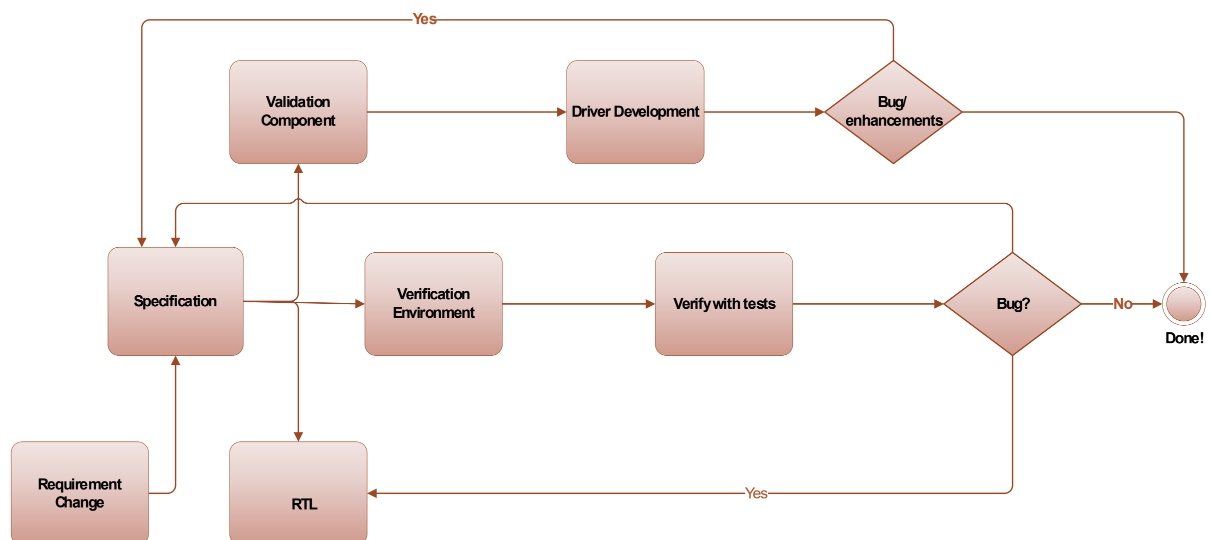


Figure 2.1: Traditional Flow

Fundamentals of Register Verification:

2.3.1 Creating a Test Plan

A solid verification plan is the foundation of effective register testing. It typically includes:

- **Reset/Default Value Tests**: Ensures each register field initialises correctly after reset.
- **Attribute Tests**: Validates access types such as Read-Only (RO), Read-Write (RW), Write-Only (WO), and Write-One-to-Clear (RW1C).
- **Functional Tests**: Confirms that register values trigger the correct hardware behaviour, such as state machine transitions or control logic.

2.3.2 Verification Methodology

The basic plan is to write a value to a register, read it back, and compare it to what you expect it to be based on the register's attributes and masks. You should test each bit at both logic levels (0 and 1). It is important to get full functional coverage, and only using directed tests can be dangerous and not very effective.

2.3.3 Efficient Test Case Design

If you write separate tests for each register, you could end up with hundreds or thousands of test cases, each of which needs full design initialization. This takes a lot of time and resources.

Combining default value and attribute checks into one test is a better way to do things. This cuts down on the number of simulations and the time it takes to run them, from days to hours. Functional tests, on the other hand, may still need separate cases for each scenario.

2.3.4 Addressing Edge Cases

Corner cases must be considered to ensure comprehensive verification. For example, if register addresses are incorrectly mapped in the design, writes and reads may target the wrong locations—an issue that's difficult to detect without thorough testing.

2.4 Literature Survey

This section presents the literature review of the research work carried out in recent years based on functional verification.

G. Sharma et al. [1] introduced a register-aware verification framework that employs Register Access Technology (RAT) in UVM settings to make it easier to check complex SoC/IP designs. Their solution is centered on making the register database available and automating the creation of testbenches to cover more functions and speed up the verification process. The results showed that the new approaches were better and faster than the old ones that used UVM and SystemVerilog.

Paul C. Attie et al. [2] came up with a new way to check quorum-based distributed register emulation algorithms when there are crashes that can't be detected. Their method uses projection and abstraction to make model checking easier by only looking at subsystems that have a set number of processes. This method solves the state-explosion problem and makes verification possible on a larger scale. The method not only checks for correctness, but it also helps find duplicate algorithms, which makes it useful for many types of fault-tolerant distributed systems.

I. Ghosh et al. [3] came up with a new design-for-verification method that is meant to improve RTL-level verification that is based on simulation. To improve simulation coverage and error detection, the method involves adding extra, well-known behavior to the circuit being verified. The extra parts are taken away after verification is successful, leaving the original design. Tests on different industrial circuits showed that this method works and has a lot of potential for making verification better.

U. Bhiogade et al. [4] looked into how to use Embedded UVM (EUVM) to improve hardware-software co-verification in SoC environments through the Register Abstraction Layer (RAL). EUVM is an open-source version of the IEEE 2020 UVM standard that works with multiple cores. It helps UVM testbenches become native binaries that can be used on embedded platforms. This method combines the best features of current UVM methods to deliver a system-level view of functional verification. The research looks at EUVM from the perspectives of RTL designers, device driver developers, and system engineers. It shows how it might help bring hardware and software testing closer together.

J. Pasagic et al. [5] talked about how to move from checking register and fortress and memory models by hand to checking them automatically crashing IP-X The paper stresses the need for standardizing this process with tools like the Magillem UVM generator, which makes verification more efficient and lowers the chance of logic and functional errors. The study also points out that industry data shows a constant specification error rate of about 45%, and that 32% of design respins are caused by changes to the specifications. The authors say that register verification should be a part of the overall device development strategy. They also stress the

importance of having verification environments that are flexible, configurable, and reproducible at all levels of abstraction.

茅乾博 et al. [6] proposed a read-only register verification test platform based on the Universal Verification Methodology (UVM). The platform includes a UVM-based test frame and a digital design module with a register and a register read-write bus interface. The UVM test sequence interacts with the design module via a UVM sequence generator and utilises the Verilog Programming Interface (VPI) to perform internal signal assignments. This method significantly improves verification efficiency and enhances confidence in the correctness of read-only register behaviour.

茅乾博. et al. [7] proposed a UVM-based write-only register verification test platform comprising a UVM test framework and a digital design module (DUT) with a register and a register read-write bus interface. The UVM test sequence interacts with the DUT via a UVM sequencer and accesses internal signals through the Verilog Programming Interface (VPI). This method enhances verification efficiency and improves the credibility of write-only register verification by enabling precise control and observation of internal DUT behaviour.

N. Kim et al. [8] came up with a formal verification-based method for checking registers that automatically makes full access policy specifications for IP memory-mapped registers. This method is different from traditional simulation-based methods that depend on manually written specifications. Instead, it uses formal techniques to come up with specifications, which are then checked by hand to make sure they match the design intent. The authors came up with a new register model that fixes the problems with how UVM and IP-XACT access policies can be too limited in what they can say. The method worked well on three industrial designs, showing that it can improve the accuracy and automation of verification.

P. A. Abdulla et al. [9] addressed the verification problem for Communicating Register Automata (BDRA), an extension of classical register automata that supports dynamic process creation. In this model, each process maintains a mailbox for message storage and a finite set of registers for storing process IDs. Processes can send messages to the mailboxes of other processes that are listed in their registers so that they can talk to each other. The major purpose of the study is to find out if there is a configuration that causes at least one process to get into an error condition. The authors explain in detail the decidable and undecidable subclasses of BDRA for different channel configurations. This helps to construct the theoretical foundations of validating distributed systems.

K. Kim et al. [10] developed a highly reliable safety-class controller for Nuclear Power Plants (NPPs) using a diversity-based approach that operates a PLC-type and a PLD-type

controller in parallel. For the PLD-type controller, structured testbenches were built using UVM classes to perform functional verification. A UVM register model was added to the testbenches to make it easier to operate and watch the DUT. This made it easy to easily inspect the data pathways between the register sets and the I/O ports. This strategy made it easier to undertake systematic constrained random verification and got rid of the necessity for hard black-box testing. The study showed that UVM register models are better because they can be used in more than one way, are easier to scale, and work with other models. It also suggested design rules for checking the safety of NPP controllers.

T. Darko et al. *[11]* talked about how standard UVM documentation doesn't do a good job of supporting bit-resolution access and complex buffer mechanisms that are needed for serial interfaces like SPI. They suggested improvements to the UVM Register Abstraction Layer that would allow for the modeling of these features while still being compatible with passive operation, which is very important for reusability in verification environments. The suggested solutions can be used with other serial bus interfaces and can be changed to deal with new problems that come up with serial access protocols.

K. Radha et al. *[12]* discussed using a Python script to automatically generate UVM-RAL (Register Abstraction Layer) models. Their approach includes built-in checks for correctness and produces complete UVM register packages. This method differs from earlier techniques that relied on Synopsys tools to convert CSV or Excel files into RAL models. The script uses register data files to make all the UVM class parts it needs, such as variables, constructors, factory registrations, tasks, and functions. This automation makes it much easier and faster to set up UVM environments from scratch. It also speeds up the verification process by quickly making full and correct models of the registers.

T. Timisescu et al. *[13]* employed the UVM Register Abstraction Layer (UVM_REG) to look into a part of SoC designs that hasn't been studied as much: memory verification. The authors showed how to employ memory sequences produced for one block in different verification environments in a vertical approach. The "front-door" method changes abstract memory operations into real bus transfers, which makes it easier to add memory sequences to multiple testbenches. This method makes it easier to use and more consistent to check memory subsystems.

2.5 Problem Formulation

Based on the literature review, the following research gaps are identified:

Modern SoC/IP designs are getting more complicated, especially when it comes to the amount of registers and the connections between them. This has made old ways of checking things obsolete. Limited observability, fragmented tool support, and manual testbench development all lead to inefficiencies, inadequate coverage, and an increased chance of functional mistakes [5].

While methodologies such as UVM and tools like IP-XACT have introduced standardisation and reusability [5], they fall short in addressing advanced verification needs, such as dynamic register behaviour [1], serial interface modelling [11], and scalable memory subsystem verification [13].

Furthermore, the lack of automation in generating UVM Register Abstraction Layer (RAL) models [12] and the limited support for formal verification techniques [8] hinder the ability to achieve high confidence in functional correctness, especially in safety-critical [10] and distributed systems [2], [9].

Most well-known solutions only things like read-only [6] or write-only [7] registermore verification, van rely on vendor-specific tools that limit flexibilityes and extensibility [12].

So, the main problem is to create and put into action a **single, automated, and scalable verification framework** that:

- **Automatically** creates UVM-based register and memory models from structured inputs (like CSV, Excel, and IP-XACT) [12]

- **Combines** Register Access Technology (RAT) and formal verification methods to make coverage and correctness better [1], [8]

- Can **handle** complex verification situations like dynamic register interactions [1], serial protocols [11], and memory reuse [13]

- UVM features such as VPI and frontdoor/backdoor access [6], [7], make it easier to see and **control** things.

- It also makes sure that things can be **reused**, **configured**, and work with a wide range of verification environments, including embedded [4], and safety-critical systems [10].

This formulation lays the groundwork for creating a strong verification solution that fixes the problems with current methods and meets the needs of next-generation VLSI design verification.

Chapter 3

Register Model and Integration Overview

3.1 Overview

Base addresses, default values, access permissions, and security settings are just a few of the things that make up register specifications. These properties often change during the design process, which makes it harder and harder to check registers. It can take a lot of time and be easy to make mistakes when you have to make changes to the whole testbench and verification environment, especially when you have to deal with hundreds of registers. This chapter talks about different kinds of registers, what they are, what they do, and how register access is handled using a register model.

3.2 Register Specification

In hardware systems, functional blocks that interface with host processors are typically controlled through memory-mapped registers. Each bit in the address space corresponds to a hardware flip-flop. Software interacts with these registers to control hardware behaviour, forming what is known as the hardware-software interface.

Registers are split into fields, and each field has a name (a mnemonic) that helps you remember it. You can get to each field in a number of ways, including:

- **Read-Only (RO)**
- **Read-Write (RW)**
- **Write-One-to-Clear (RW1C)**

Some fields or registers may be set aside for later use or not used at all.

Registers are generally categorised into:

- **Configuration Registers**: The host can get to them.
- **Internal Registers**: These are used inside the IP and the host can't see them.

There are things like offset, name, size, and description for each register. Lock mechanisms, shadow registers, and hierarchical paths are some of the more advanced features. There are blocks of registers, and each block has its own base address and metadata.

Table 3.1: Attributes Overview

Field Attributes	**Register Attributes**	**Block Attributes**
Description	Description	Description
Name	Name	Name
Starting Bit	Offset Address	Base Address
Size	Reset Value	—
Access Type	Width	—

Register specifications are often maintained in spreadsheets and later converted for automation.

3.3 Register Model Use Flow

A register model is a way to use software to show control registers and memory in a verification environment. The Register Abstraction Layer (RAL) idea is what most people use to make it.

The RAL model:

- Includes predefined tests and optional functional coverage.
- Supports both front-door and backdoor access.
- Enables automation of register verification in large, complex designs.

Once generated, the model can be compiled with other testbench components. Any changes in the register specification only require re-running the generation script—manual updates are not needed.

Registers are accessed by name using RAL methods, ensuring that the environment stays synchronised with the latest specification.

3.4 Register Model Integration Overview

To add a register model to a testbench, you need the following parts:

- **Adapter:** Changes RAL transactions into sequence items that can be used on the bus.
- **Bus Agent:** Sends these sequence items to the DUT through a driver.

- **Predictor:** watches bus transactions and makes changes to the register model as needed.
- **Register Model:** A group of SystemVerilog classes that show the register map.

The adapter reads the RAL data and makes transactions. The bus agent carries out these transactions, and the predictor makes sure that the model shows the current state of the hardware.

Table 3.2: Integration Components

Component	Description
Register Model	SystemVerilog classes representing the register map.
Adapter	Converts register operations into bus-compatible transactions.
Bus Agent	Sends transactions to the DUT via a driver.
Predictor	Updates the register model based on observed bus activity.

This integration ensures that the verification environment remains consistent, scalable, and maintainable, even as the design evolves.

Chapter 4

Implementation of Register Abstraction Layer (RAL)

This chapter outlines the practical application of the Register Abstraction Layer (RAL) in the context of register verification. It details the integration process, the components involved, and how register access is managed within a verification environment.

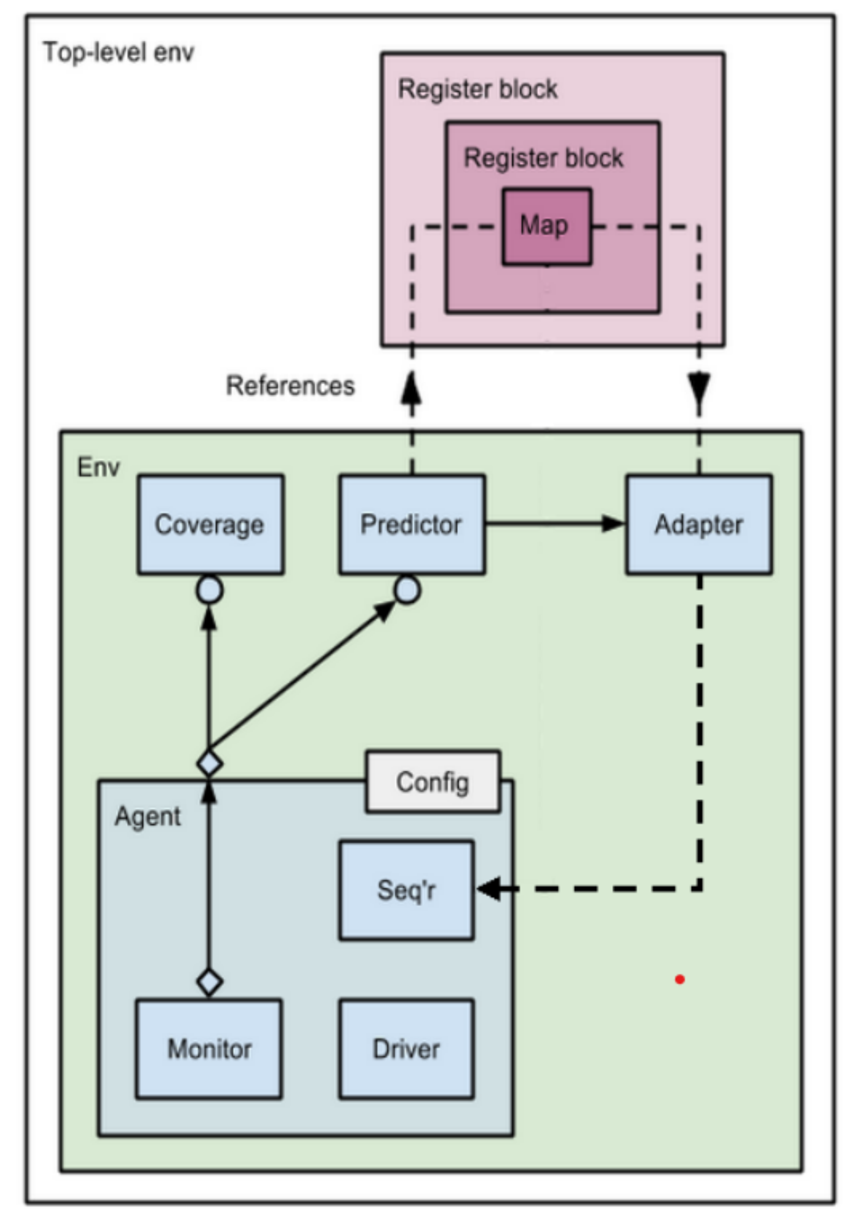


Figure 4.1: Register Abstraction Layer

4.1 Adapter

The adapter translates between general register operations and the specific bus protocol used in the design. In UVM-based environments, this is usually done by adding to the uvm_reg_adapter class. The adapter changes transactions at a high level into sequence items at a bus level and the other way around.

In this project, the adapter only works in one direction. It gets the register name from the RAL model, finds the right register object in an internal associative array, and gets its address. The adapter chooses the right agent to handle the transaction based on a user-defined attribute called access_path. The transaction is then limited and sent to the bus agent to be carried out.

4.2 Bus Agent

The bus agent is a standard verification part that has a sequencer, a driver, and a monitor. Its major task is to link to the DUT by making and keeping track of transactions at the pin level. In this setup, the adapter initiates a short sequence that runs via the sequencer and then to the driver. After that, the driver either reads or writes to the bus. For read operations, the testbench gets the result back.

4.3 Predictor

The RAL model keeps a mirrored copy of each register's value that is updated with each transaction. You can change this mirror by hand using the predict and update methods, or you can let a monitor do it for you. The monitor keeps an eye on bus-level activity and makes changes to the mirror as needed.

In systems with more than one master, manual updates might not pick up changes made by other agents. In these situations, it's better to use a monitor for automatic updates. For systems with only one master, manual updates are usually enough.

4.4 Front Door and Back Door Access

Registers can be accessed in two primary ways:

- **Front Door Access:** Involves using the standard bus interface, which consumes simulation cycles and adheres to protocol timing.

- **Back door Access:** Bypasses the bus and directly accesses the register using its HDL path, allowing for faster configuration and verification.

Backdoor access is especially useful for initialising large numbers of registers quickly and for verifying write-only registers or detecting address decode issues.

4.5 Test and Test Sequence

You can set up the test environment during the build phase so that certain parts can be turned on or off. In the run phase, RAL base class methods are used to access registers and run user-defined sequences.

A normal test flow looks like this:

1. Checking the reset value of a register by reading it.
2. Putting down a new value.
3. Reading again to make sure the write and access attributes are correct.

Functional verification tests are done separately, but default value and attribute checks are often done together to save time.

4.6 Handling Special Cases

4.6.1 Lock Bits

Some registers have fields that depend on the state of other registers—these are known as lock bits. Handling them requires custom logic, often implemented in the base class using user-defined attributes. This approach simplifies test development and ensures consistent behaviour.

4.6.2 Write-Only Registers

You can't check write-only registers with front-door reads because they always return zero. Instead, backdoor access is used to make sure that the write was done right. In the RDL, you need to define the register's hierarchical path for this to work. Writing through the front door and reading through the back door makes sure everything is correct and saves time on the simulation.

4.6.3 Special Bits

Test sequences shouldn't include things like "reset" or "enable." Writing to these bits can reset the design or turn off parts of the system, which can mess up the verification process. Other edge cases are sequences that interact with registers during initialization, which need to be handled with care.

4.7 Implementation Algorithms

4.7.1 Register Verification Flow

Input: User-defined register verification sequence
Output: Register-level pass/fail status

1. **Begin**
2. Initialize or configure the test environment, if required
3. Invoke the user-defined register verification sequence
    a. Apply stimulus
    b. Monitor register behaviour
    c. Capture response and check against expected values
4. Log results (pass/fail)
5. **End**

4.7.2 Register Attribute Verification Sequence

Input: Top-level RAL model
Output: Pass/fail status for register field attribute verification across all sub-IPs

1. **Begin**
2. Obtain pointer to the top-level RAL class
3. For each register file in the RAL hierarchy:
    a. Retrieve register file pointer
    b. For each register in the file:
        i. Retrieve register pointer
        ii. Perform reset value check
        iii. Perform attribute check:
            - Update RAL model if necessary

- Generate appropriate write data
- Write data to register
- Read back and verify against expected value
- Apply RW1C protection logic to avoid unintentional bit clearing

4. Repeat steps 3a–3b for all register files across 15 sub-Ips
5. Log overall test results (per sub-IP)
6. **End**

4.7.3 RAL Access Adapter Sequence Algorithm

Input:

- reg_name: Name of the target register
- data: Data to be written (if applicable)
- op_type: Operation type (e.g., READ, WRITE)

Output:

- Read data (if op_type == READ)
- Pass/fail status or transaction completion indicator

1. **Begin**
2. Use user-defined utility to retrieve physical address of reg_name
3. Based on op_type, set opcode and direction
4. Constrain the sequence object with:
    a. Register address
    b. Data (for WRITE)
    c. Opcode and access path
5. Launch the sequence on the corresponding sequencer (e.g., APB/AXI)
6. If op_type is READ, return the received data
7. **End**

Chapter 5

WORK DONE AND RESULT DISCUSSION

5.1 Test Plans

The table below outlines the various test scenarios used to verify the functionality of the Registers. These tests ensure that both typical and edge cases are covered during verification.

Table 5.1: Test Plan

S. No.	Verification Objective	Test Method
1.	All registers have the correct reset value per the register specification.	Read the registers and compare returned data against the default value in RAL.
2.	Registers can be written and read successfully, and don't impact other registers in the same IP.	Read-Write-Read each register in the IP in two different orders.
3.	FUB's common code for accessing the address space works.	Read-Write-Read one register on each FUB using each supported address space.
4.	FUB's common interface code works across each address space.	Read-Write-Read one register on each FUB on each HW-accessible interface using each address space. Includes illegal values to confirm they behave appropriately.
5.	Ensures back-to-back reads work correctly.	For each FUB, Read-Read-Write-Write-Read-Read & reads must happen in immediately adjacent bus cycles.
6.	Ensures back-to-back writes work correctly.	For each FUB, Read-Read-Write-Write-Read-Read & writes must happen in immediately adjacent bus cycles.

Table 5.2: Different modes for the Attribute test

MODE: 0	Write one register at a time, followed by read and check.
MODE: 1	Write all registers, followed by read and check for all.
MODE: 2	Write one register at a time, followed by read and check, then read and check all registers.
MODE: 3	Register alias testing. write one register at a time, followed by read_and_check all registers.

5.2 Register Testing Summary

5.2.1 Reset Value Verification

After asserting a reset, all registers were read, and their values were compared against the default values defined in the RAL model. This confirmed that the DUT correctly initialises all registers as per the specification.

5.2.2 Attribute Register Access – Mode-Based Testing (Modes 0–3)

Each register was written with a test value and then read back to verify data integrity. This was repeated across four attribute modes (Mode_0 to Mode_3) to ensure consistent behaviour. The test confirmed that registers are writable, readable, and isolated from each other.

5.2.3 Register Access Order Independence

Registers were accessed in different sequences (e.g., out-of-order Read-Write-Read) to validate that register operations are order-independent and do not interfere with one another. This ensured robustness in access patterns.

5.2.4 Address Space Access Validation

One register per FUB was accessed using each supported address space. Read-Write-Read operations were performed to confirm that the common code correctly handles address decoding and access routing across all address spaces.

5.2.5 Interface-Level Access and Error Handling

Registers were accessed through all hardware-accessible interfaces using each address space. Illegal values were also injected to verify that the design handles invalid accesses gracefully. This confirmed interface-level robustness and error handling.

5.2.6 Back-to-Back Read Operation Verification

A sequence of Read-Read-Write-Write-Read-Read was executed with the read operations occurring in immediately adjacent bus cycles. This tested the DUT’s ability to handle consecutive reads without timing or data issues.

5.2.7 Back-to-Back Write Operation Verification

Similar to the previous test, a Read-Read-Write-Write-Read-Read sequence was used, but the focus was on verifying that two consecutive write operations in adjacent bus cycles are handled correctly. This validated timing compliance and data integrity under high-throughput conditions.

5.3 Tools Used in Verification:

5.3.1 ***Synopsys VCS*** is a popular verification tool that has fast simulation engines and full support for System Verilog. It uses Fine-Grained Parallelism (FGP) to run tests faster on multiple cores. It has a number of new features, such as Synopsys Design Constraints (SDC) verification, Intelligent Coverage Optimization (ICO), and Dynamic Test Loading (DTL). These features make it easier to find bugs early in the design process.

Key benefits:

- Industry-standard simulation engine
- Broad System Verilog support
- Built-in support for coverage analysis and debug tools

5.3.2 ***Synopsys Verdi*** is a powerful debugging tool that makes it easier for design and verification engineers to understand complicated designs. It has automated tools for tracing behavior and finding errors, which cuts down on debugging time by 50% or more. Verdi has a full-featured waveform viewer, a source code browser, and a waveform comparison engine.

Key features:

- Automates tracking and behaviour analysis
- Flexible logic views for enhanced debugging
- Saves time in complex debugging tasks

5.4 Result and Wave

5.4.1 Waveforms confirm that, following reset, all registers return their default values as specified. Read operations retrieve data matching the expected reset values defined in the Register Abstraction Layer (RAL), verifying correct initialisation behaviour.

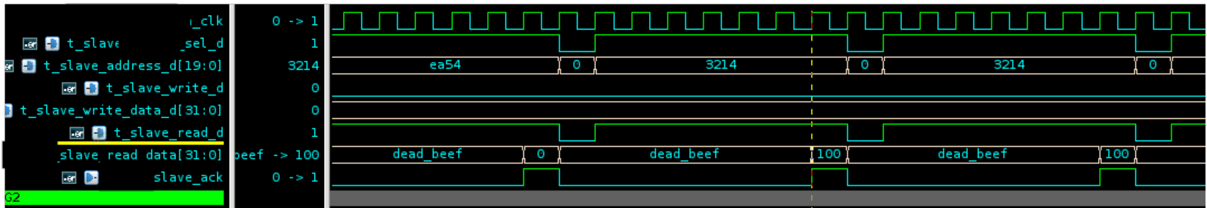


Figure 5.1: Reset Test

5.4.2 Waveforms demonstrate that all registers within the IP can be written to and read back successfully, with no unintended interaction between registers. Read-Write-Read operations confirm data integrity and register independence.

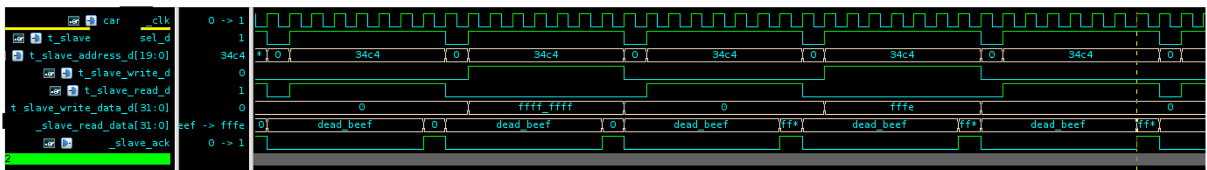


Figure 5.2: Attribute Test Mode_0

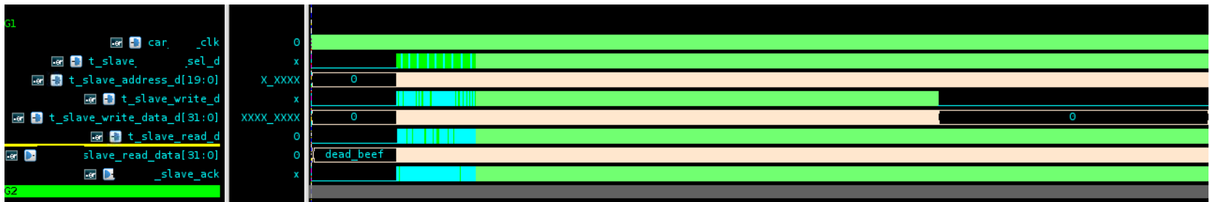


Figure 5.3: Attribute Test Mode_1

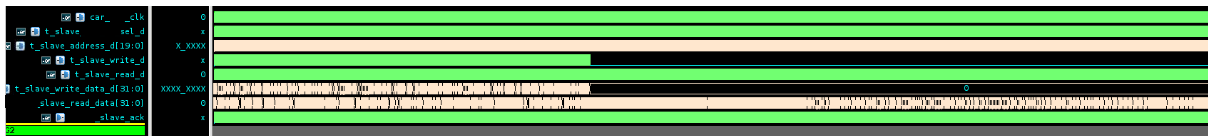


Figure 5.4: Attribute Test Mode_2

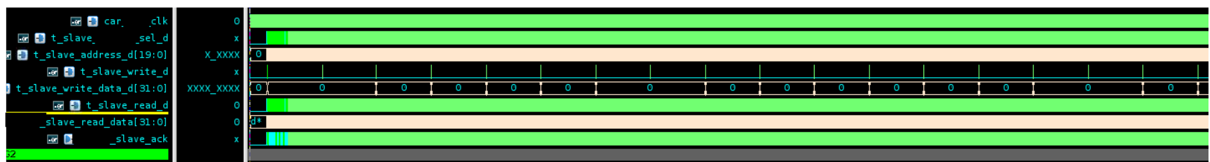


Figure 5.5: Attribute Test Mode_3

5.4.3 Waveforms demonstrate that all registers within the IP can be written to and read back successfully in varying sequences, with no unintended interaction between registers. Read-Write-Read operations confirm data integrity and register independence.

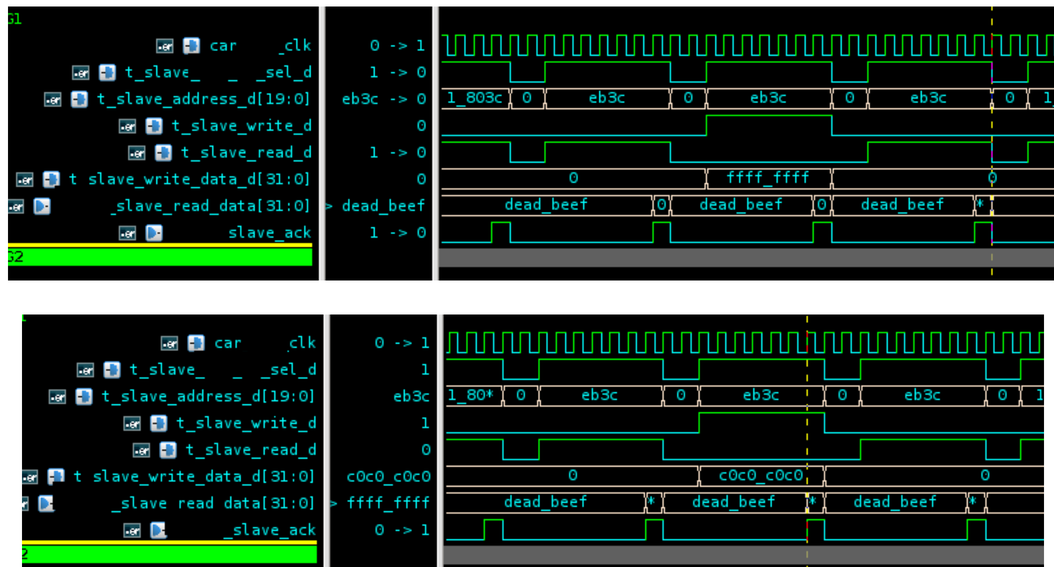


Figure 5.6: Varying sequences test (2.1)

5.4.4 *Waveforms confirm that the common code for accessing FUB address spaces functions correctly. Read-Write-Read operations on one register per FUB across all supported address spaces verify consistent and reliable access.*

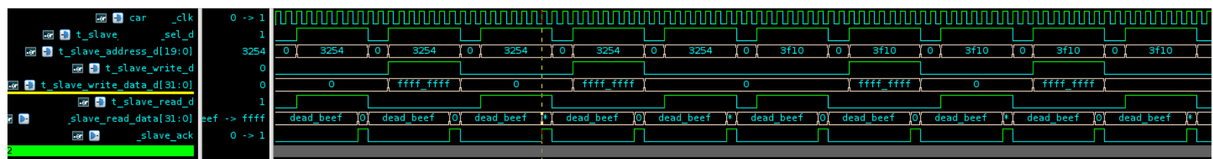


Figure 5.7: Address spaces test (2.2)

5.4.5 Waveforms validate that the FUB's common interface code operates correctly across all hardware-accessible interfaces and supported address spaces. Read-Write-Read operations on one register per FUB confirm consistent access, while tests with illegal values demonstrate appropriate error handling and robustness.

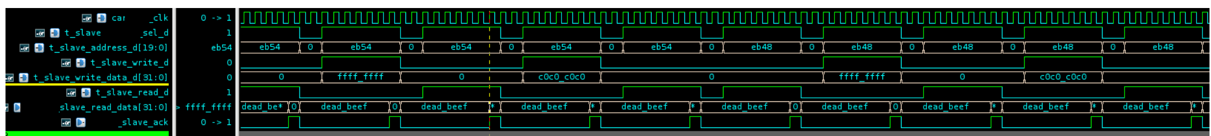


Figure 5.8: Accessible interfaces test (2.3)

5.4.6 Waveforms confirm that back-to-back read register accesses function correctly across all FUBs. Sequences of Read-Read-Write-Write-Read-Read executed in immediately adjacent bus cycles demonstrate reliable data handling and interface timing compliance.

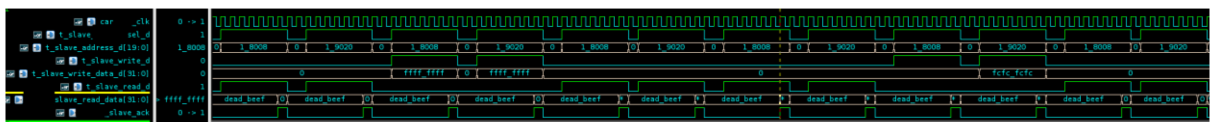


Figure 5.9: B2B Read Test (2.12)

5.4.7 Waveforms verify that back-to-back write operations function correctly across all FUBs. In Read-Read-Write-Write-Read-Read sequences, the two write transactions occur in immediately adjacent bus cycles, confirming proper timing and data integrity under consecutive access conditions.

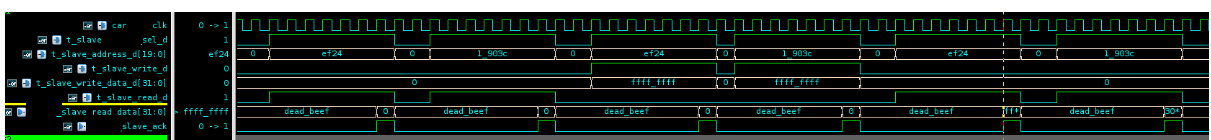


Figure 5.10: B2B Write Test (2.11)

5.5 Design Bugs and Solutions

During the course of Register verification, several design issues were identified through waveform analysis and UVM-based testing. These bugs impacted register behaviour, signal integrity, and interface consistency. Each issue was carefully analysed, reported to the relevant design teams, and resolved through RTL updates or register model corrections. The table below summarises the key bugs encountered, their root causes, and the solutions implemented to ensure design correctness and compliance with specifications.

Table 5.3: Summary of Identified Design Bugs and Implemented Solutions

Bug ID	Description	Impact	Root Cause	Proposed Solution/ Workaround
DB-001	Register returns incorrect default value after reset	Incorrect initialisation behaviour	Reset logic did not properly propagate to all registers	Updated reset synchronisation logic in RTL / Excluded
DB-002	Back-to-back writes occasionally fail on the FUB interface	Data corruption in consecutive writes	Write enable signal not held long enough during back-to-back cycles	Extended writes enable duration by one cycle / Excluded
DB-003	Illegal address access does not trigger the expected error response	Potential for undefined behaviour	Address decoder missing illegal address check	Added illegal address detection logic
DB-004	Read-after-write returns stale data in some registers	Data inconsistency	Write completion is not synchronized with the read path	Inserted read-after-write hazard handling logic
DB-005	Register read mismatch: DUT value 0xFFFFFFFF vs. mirrored value 0x0000FFFF	UVM register model mismatch caused test failures and false error reports	Inconsistent register width or partial update not reflected in the mirror	The register model was updated to match DUT behaviour; mirror synchronisation logic was corrected

```
UVM_ERROR /u/s/reg/uvm_reg.svh(314) @ 5740.800000 ns: reporter [RegModel] Register "ral_top.top.port_0.T1_inst[18]" value read from DUT (0x00000000f8ffffff) does not match mirrored va
lue (0x00000000167f7f7f)

UVM_ERROR /u/s/reg/uvm_reg.svh(314) @ 9196.800000 ns: reporter [RegModel] Register "ral_top.top.port_0.T2_inst[23]" value read from DUT (0x00000000f8ffffff) does not match mirrored va
lue (0x00000000167f7f7f)

UVM_ERROR /u/s/reg/uvm_reg.svh(314) @ 9542.400000 ns: reporter [RegModel] Register "ral_top.top.port_0.T3_inst[116]" value read from DUT (0x00000000f8ffffff) does not match mirrored v
alue (0x00000000167f7f7f)

UVM_ERROR /u/s/reg/uvm_reg.svh(314) @ 9657.600000 ns: reporter [RegModel] Register "ral_top.top.port_0.T4_inst[108]" value read from DUT (0x00000000f8ffffff) does not match mirrored v
alue (0x00000000167f7f7f)
```

Figure 5.11: Error Description

UVM_ERROR/u/s/reg/uvm_re.svh(314)@40057600:reporter[RegModel]Register"ral_top.top.port_0.T_inst"value read from DUT (0x00000000ffffffff) does not match mirrored value (0x000000000000ffff)

Chapter 6

CONCLUSION AND FUTURE SCOPE

6.1 Conclusion

Implementing a register model significantly streamlines the traditionally error-prone and time-intensive process of register design and verification. This approach offers a low-maintenance, reusable framework that enables design, verification, and firmware teams to collaborate more effectively using a unified and synchronised view of the chip architecture.

The method described in this paper can be used with different IP blocks, SoCs, and even at the chip level. By using this model-based flow, teams can cut down on a lot of work and time, which results in designs that are more stable and have been thoroughly tested. In real life, things that usually take more than a month of manual work can be done in less than a week, and tests can be done in less time. The Register Abstraction Layer (RAL) method not only speeds up the process of checking IP, but it also improves it and gets it to market faster.

6.2 Future Scope

Backdoor Access Optimisation, *once front-door access to registers is verified, backdoor access can be utilised* to accelerate subsequent verification tasks. This method allows for faster register manipulation, especially in large-scale simulations, and helps reduce overall test execution time.

Advanced Monitoring Mechanisms, in complex IPs where multiple agents may access or modify registers—either through hardware logic, strapping, or forced values—a dedicated monitor can be implemented. This monitor keeps a synchronized mirror of the register state by using backdoor access. It also lets you check things like write-only (W/O) registers, which can't be checked with normal read operations.

Enhanced Coverage Analytics, Basic coverage is already in place. Future improvements could include more detailed analytics, such as tracking per-bit toggles, cross-coverage between register fields and functional states, and integration with formal coverage tools. These improvements would give us more information about how complete our verification is and help us find scenarios that haven't been tested yet.

REFERENCES

1. G. Sharma, L. Bhargava and V. Kumar, "Automated Coverage Register Access Technology on UVM Framework for Advanced Verification," 2018 IEEE International Symposium on Circuits and Systems (ISCAS), Florence, Italy, 2018, pp. 1-4, doi: 10.1109/ISCAS.2018.8351413.

2. Attie, Paul C., and Chockler, Hana (2006). Automatic Verification of Fault-Tolerant Register Emulations, Electronic Notes in Theoretical Computer Science, 149(1), 49–60.

3. I. Ghosh, K. Sekar and V. Boppana, "Design for verification at the register transfer level," Proceedings of ASP-DAC/VLSI Design 2002. 7th Asia and South Pacific Design Automation Conference and 15h International Conference on VLSI Design, Bangalore, India, 2002, pp. 420-425, doi: 10.1109/ASPDAC.2002.994957.

4. Bhiogade, Utkarsh, Joshi, Kautilya, and Goel, Puneet (2021). System-Level Register Verification and Debug. Proceedings of DVCon Europe [2021: Munich, Germany: 2021], pp. 1–6.

5. Pasagic, Jasminka, and Donner, Frank (2017). SimpleLink™ MCU Platform: IP-XACT to UVM Register Model – Standardising IP and SoC Register Verification. Proceedings of DVCon Europe [2017: Munich, Germany: 2017], pp. 1–6.

6. 茅乾博. Read-only register verification test platform and verification method based on UVM (Universal Verification Methodology Manual). Technical Report: China Patent CN105446844A, 2014, pp. 1–10.

7. 茅乾博. UVM (Universal Verification Methodology) based write-only register verification test platform and verification method. Technical Report: China Patent CN105320583A, 2014, pp. 1–10.

8. Kim, NamDo (2013). Register Verification: Do We Have Reliable Specification? Proceedings of DVCon [2013: San Jose, USA: 2013], pp. 1–6.

9. Abdulla, P.A., Atig, M.F., Kara, A., and Rezine, O. Verification of Buffered Dynamic Register Automata. In Bouajjani, A., and Fauconnier, H. (Eds.),

NETWORKED SYSTEMS. NETYS 2015. LECTURE NOTES IN COMPUTER SCIENCE, VOL. 9466. CHAM: SPRINGER, 2015, PP. 15–30.

10. KIM, KYUCHULL (2014). FUNCTIONAL VERIFICATION OF A SAFETY CLASS CONTROLLER FOR NPPS USING A UVM REGISTER MODEL, NUCLEAR ENGINEERING AND TECHNOLOGY, 46(3), 381–386.

11. TOMUŠILOVIĆ, DARKO M. (2016). EXTENDING UVM REGISTER ABSTRACTION LAYER FOR VERIFICATION OF REGISTER ACCESS VIA SERIAL BUS INTERFACE. PROCEEDINGS OF DVCON EUROPE [2016: MUNICH, GERMANY: 2016], PP. 1–6.

12. SURENDRA KUMAR K., V.S., AND RADHA, K. (2021). A PYTHON AUTOMATION SCRIPT THAT GENERATES UVM_RAL (REGISTER ABSTRACTION LAYER) REGISTER MODEL, MATERIALS TODAY: PROCEEDINGS, 37(2), 2233–2235.

13. TIMISESCU, TUDOR (2015). LEVERAGING THE UVM REGISTER ABSTRACTION LAYER FOR MEMORY SUB-SYSTEM VERIFICATION IMPLEMENTING MEMORY SEQUENCE REUSE ACROSS MULTIPLE UNDERLYING BUS PROTOCOLS. PROCEEDINGS OF DVCON EUROPE [2015: MUNICH, GERMANY: 2015], PP. 1–6.

14. SYSTEMVERILOG FOR VERIFICATION A GUIDE TO LEARNING THE TESTBENCH LANGUAGE FEATURES - CHRIS SPEAR.

15. ASIC/SOC FUNCTIONAL DESIGN VERIFICATION, A COMPREHENSIVE GUIDE TO TECHNOLOGIES AND METHODOLOGIES - ASHOK B. MEHTA.