Automotive grade IP and SoC Advanced Verification

Project report submitted in partial fulfilment of the requirement for the Award of the Degree of

MASTER OF ENGINEERING

in ELECTRONICS AND COMUNICATION ENGINEERING

Submitted By

Kapil Joshi

802361003

Under Supervision of

Dr. Geetika Dua

Assistant Professor

&

Dr. Kulbir Singh

Professor and Head, Associate Dean

&

Saransh Mehrotra

Manager

At

ST Microelectronics

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

THAPAR INSTITIUTE OF ENGINEERING AND TECHNOLOGY

(A DEEMED TO BE UNIVERSITY), PATIALA, PUNJAB

JULY 2025

CERTIFICATE

This is to Certify that project entitled " Automotive grade IP and SoC Advanced Verification" which is being submitted by Kapil Joshi (University Registration No. 802361003) to the Department of Electronics and Communication Engineering, TIET, Patiala, Punjab, is a record of project work carried out by him under guidance and supervision of Mr. Saransh Mehrotra (Principal Engineer, ADG, STMicroelectronics India Pvt. LTD), during a period from 12 June 2024 to 31 July 2025. This is to certify that the candidate's above statement is correct to the best of my knowledge.

Saransh Mehrotra

Manager, Principal Engineer
STMicroelectronics Pvt. Ltd.,
Greater Noida, India

DECLARATION

I hereby declare that the project work entitled is "Automotive grade IP and SoC Advanced Verification" in partial fulfilment of the requirement of the award of the degree of Master of Engineering (M.E ECE) submitted at Electronics and Communication Engineering Department, Thapar institute of Engineering & Technology(Deemed to be university), Patiala is a record of work carried out under supervision of **Mr. Saransh Mehrotra** (Principal Engineer, STMicroelectronics), **Dr. Geetika Dua** (Assistant Professor, Electronics and Communication Engineering Department, Thapar Institute of Engineering & Technology(Deemed to be university) and **Dr. Kulbir Singh** (Professor and Head, Associate Dean, Electronics and Communication Engineering Department, Thapar Institute of Engineering & Technology (Deemed to be university) from June 2024 to July 2025. The matter in this has not been submitted in part to any other university or institute for the award of any other degree.

Certified that the above statement made by the student is correct to the best of our knowledge and belief.

Kapil Joshi	Dr. Geetika Dua	Dr. Kulbir Singh
802361003	Academic Supervisor Assistant Professor, ECED, TIET, Patiala, India	Academic Supervisor Professor and Associate Dean ECED, TIET, Patiala, India

ACKNOWLEDGEMENT

I would like to express my gratitude to my college mentor, **Dr. Geetika Dua** and **Dr. Kulbir Singh** for their continuous support and guidance throughout my internship. I would also like express my gratitude to my industry mentor, **Saransh Mehrotra** and **Abhishek Kapoor**. Their advice, wisdom, and encouragement have been invaluable in helping me navigate through the challenges and make the most out of this experience. I am thankful for his unwavering support and for being a constant source of motivation. Their guidance, support, and encouragement have been invaluable, and I am truly grateful for their contributions.

I am deeply grateful to **Dr. Kulbir Singh**, Head of the Department (ECE) and **Dr. Amit Mishra**, M.E (ECE) Coordinator, Thapar Institute of Engineering. & Technology, Patiala for providing me with a learning Environment and Infrastructure in ECED.

I will be ignorant if I do not express my gratitude to the author of the references and other literature cited in this project.
Finally, I want to thank all my colleagues for their encouragement through potential discussions and suggestions.

ABSTRACT

Adding complicated electronic systems to cars is making them safer, better at what they do, and easier to use. This is a time of big change for the car industry. It is very important to make sure that these systems are safe to use so that disasters don't happen as they get more complicated. The International Organization for Standardization (ISO) 26262 is a set of rules that says how safe car systems must be. It gives rules and guidelines to make sure that these systems are safe and dependable. It is very hard to meet the requirements of ISO 26262, especially when it comes to checking complex System-on-Chip (SoC) architectures and Intellectual Property (IP) components. The NVMPC IP has to be created because automotive systems are becoming more complicated, connected and autonomous vehicles are becoming more common, and strict functional safety standards must be met. The NVMPC IP has extensive memory management features like error detection and repair, data caching, and protocol translation to make sure that data is safe and reliable. Its development meets the growing demand for strong memory management solutions in today's car systems.

The NVMPC IP needs to be made because cars are getting more complex, connected, and self-driving cars are becoming more common. It also needs to meet strict functional safety standards. The NVMPC IP has a lot of memory management tools, such as error detection and repair, data caching, and protocol translation, to make sure that data is safe and reliable. Its growth meets the need for better memory management in today's car systems.

Table of Contents

ABSTRACT v

List of Figures ix

List of Abbreviations x

Chapter 1 : Introduction 1

Chapter 2 : Background 2

2.1 ISO 26262 Standard 2

2.2 Importance of ISO 26262 in NVMPC Verification 3

2.3 Conclusion 4

Chapter 3 : Literature Review 5

3.1 Literature section 5

3.2 Research gaps 8

Chapter 4 : Problem Statement 9

Chapter 5 : IP Verification Methodology 10

5.1 Verification Methodolgy 11

5.2 SV Based Methodology System 11

5.3 UVM Based Methodology 11

5.4 Assertion Based Methodology 12

Chapter 6 : Work done 13

6.1 Blocks and components: 13

6.2 Flow of AXI signals 14

6.3 Testbench structure 15

6.4 Verification planning 15

6.5 UVM Components 16

6.5.1 UVM Sequence and sequence item 16

6.5.2 UVM Sequencer 17

6.5.3 UVM Driver 17

6.5.4 UVM Monitor 18

6.5.5 UVM Agent 18

6.5.6 UVM Scoreboard 19

6.5.7 UVM Environment 19

6.5.8 UVM Test 19

6.5.9 UVM Top..20

6.6 UVM Phases 20

6.6.1 Build phase 20

6.6.2 Connect phase 21

6.6.3 Run phase..21

6.6.4 Extract phase 22

6.7 Output waveform 22

6.8 Observations: 23

6.9 Testcases..24

6.9.1 Basic test sequence 24

6.9.2 Reset value check test 24

6.9.3 Reg access test..25

6.9.4 Endian check test 25

6.10 Simulation results: 26

6.11 Regression report 26

6.12 Coverage Report..27

6.13 Summary 28

Chapter 7 : SoC (System on Chip) Methodology 29

7.1 System on Chip (SoC) Design Flow 29

7.1.1 Requirements..30

7.1.2 Feasibility..30

7.1.3 Architecture Design..30

7.1.4 Frontend Design..30

7.1.5 Verification..30

7.1.6 Design for Testability..31

7.1.7 Backend Design 31

7.1.8 Tapeout..31

7.1.9 Fabrication 31

7.1.10 Testing and Validation 31
7.1.11 Product Release 32
7.1.12 Post–Silicon Validation 32
Chapter 8 : Conclusion 33
Chapter 9 : Future scope 34
References 35
Appendix A : Plagarism report 38

List of Figures

Figure 1: ISO26262 logo 2
Figure 2: ISO26262 Block Diagram 3
Figure 3: Verification management flow 10
Figure 4: SV Testbench 11
Figure 5: UVM Testbench 12
Figure 6: NVMPC UVM Environment 13
Figure 7: Verisium manager tool 16
Figure 8: vPlan structure 16
Figure 9: UVM Sequence item 17
Figure 10: UVM Sequencer 17
Figure 11: UVM Driver 18
Figure 12: UVM Monitor 18
Figure 13: UVM Agent 18
Figure 14: UVM Scoreboard 19
Figure 15: UVM Environment 19
Figure 16: UVM top 20
Figure 17: UVM Build phase 20
Figure 18: UVM connect phase 21
Figure 19: UVM Run phase 21
Figure 20: UVM Extract phase 22
Figure 21: Waveform protocol signal flow 22
Figure 22: Basic test 24
Figure 23: Reset value check test 24
Figure 24: Register access test 25
Figure 25: Endian check test 25
Figure 26: PASS criteria 26
Figure 27: UVM Report summary 26
Figure 28: Regression results 27
Figure 29: Coverage report 100% 27
Figure 30: Code coverage report 28
Figure 31: SoC Design Flow 29

List of Abbreviations

NVM	Non-Volatile Memory
NVMPC	Non-Volatile Memory Platform Controller
EEPROM	Electrically Erasable Programmable Read-Only Memory
IP	Intellectual Property
SoC	System on Chip
ISO	International Organization for Standardization
HARA	Hazard Analysis and Risk Assessment
ASTI	Automotive Safety Test Interface
ASIL	Automotive Safety Integrity Level
ASIC	Application-Specific Integrated Circuit
UVM	Universal Verification Methodology
SV	System Verilog
DUT	Design Under Test
NAND	Not AND
SSD	Solid State Drive
API	Application Programming Interface
AXI	Advanced eXtensible Interface
UVC	Universal Verification Component

Chapter 1: Introduction

Because it can store data even when it is not connected to a power supply, non-volatile memory, also known as NVM, stands out among all other memory standards. One of the most notable characteristics of flash memory, which is a type of non-volatile memory, is that it has a relatively cheap cost per access in the current market. EEPROMs, which stand for electronically erasable programmable read-only memories, are a type of non-volatile memory (NVM) that allows for the electrical programming and erasing of the contents of the memory. When it comes to memory, flash memory is classified based on its capacity to delete complete memory blocks. Flash memories have a higher density, which means that more memory can fit in a given area of silicon. Flash memories also let you read things faster. On the other hand, they can only last for so long because of the number of read-write operations they can handle. This makes it clear that flash memory is mostly used for program memory, while other EEPROMs are used for data memory.[1] The Non-Volatile Memory Platform Controller (NVMPC) IP is a crucial component in automotive applications. The NVMPC IP ensures that memory operations are executed efficiently and reliably by managing the interface between the host system and various types of NVM. This Internet Protocol (IP) must be configurable by users to allow them to select how the program utilises the network virtual machine (NVM).[1]

A dual-port buffer stores all the data that is read or written to the NVM. It can be accessed from either side. The NVMPC infrastructure takes care of this buffer. The controller's address translator links the real address of the NVM to the virtual address of the host device. It is not necessary for the host to be aware of the particular physical address in order to accomplish reading, writing, or erasing data from the NVM. Mapping is the reason behind this. The reason for this is that it makes the process a great deal simpler. While the data is being communicated from the host to the NMM, there are several things that could go wrong, which would result in the information being distorted. Some of these issues are associated with the transmission path, while others are associated with clock jitter and a variety of different forms of coupling concerns. There is a technique for error correction included into the NVMPC IP. [1]

One of the requirements that the NVMPC IP was made to meet is the functional safety standards set by 1S0 26262. This is so that it can meet the strict needs of the automotive industry. Additionally, this makes sure that the NVMPC IP not only does what it was made to do correctly, but also helps make mechanical systems in cars safer and more reliable overall. The NVMPC IP is very important for making sure that memory processes in automotive applications are real and work well by controlling the interface between the host and the NVM. The control of the interface makes this possible.

Chapter 2: Background

2.1 ISO 26262 Standard

The International Organisation for Standardisation (ISO) 26262 is a global standard that ensures the electric and electronic systems in production automobiles are safe to use. In order to ensure that automotive systems are created, constructed, and tested in accordance with stringent safety standards, it establishes the regulations that must be followed. Because of this, the likelihood that they will fail and produce potentially hazardous circumstances is reduced.[3]

Figure 1 ISO26262 logo

Key Aspects of ISO 26262:

Safety Lifecycle: The entirety of the safety lifespan of automotive systems is encompassed by ISO 26262, beginning with the concept stage and continuing through product development, manufacturing, operation, servicing, and decommissioning. Doing a Risk Assessment and Hazard Analysis: The standard puts a lot of weight on hazard analysis and risk assessment (HARA), which is a way to find possible dangers and figure out how risky they are, when it comes to figuring out the Automotive Safety Integrity Level (ASTI) that each part of the system needs to have.[4]

ASIL Levels: ASILs go from ASIL A, which means the least amount of safety is needed, to ASIL D, which means the most safety is needed. As the ASILs move forward, the safety measures and verification activities that are required are becoming more and more strict.

Verification and Validation: ISO 26262 requires strict verification and validation procedures to make sure that all safety standards are met. Some of the methods that fall under this area are formal verification, fault injection testing, and coverage analysis. Keeping track of documents, i.e to comply with the standard, it is required to have full documentation and traceability of the safety requirements, design decisions, verification methods, and test findings.

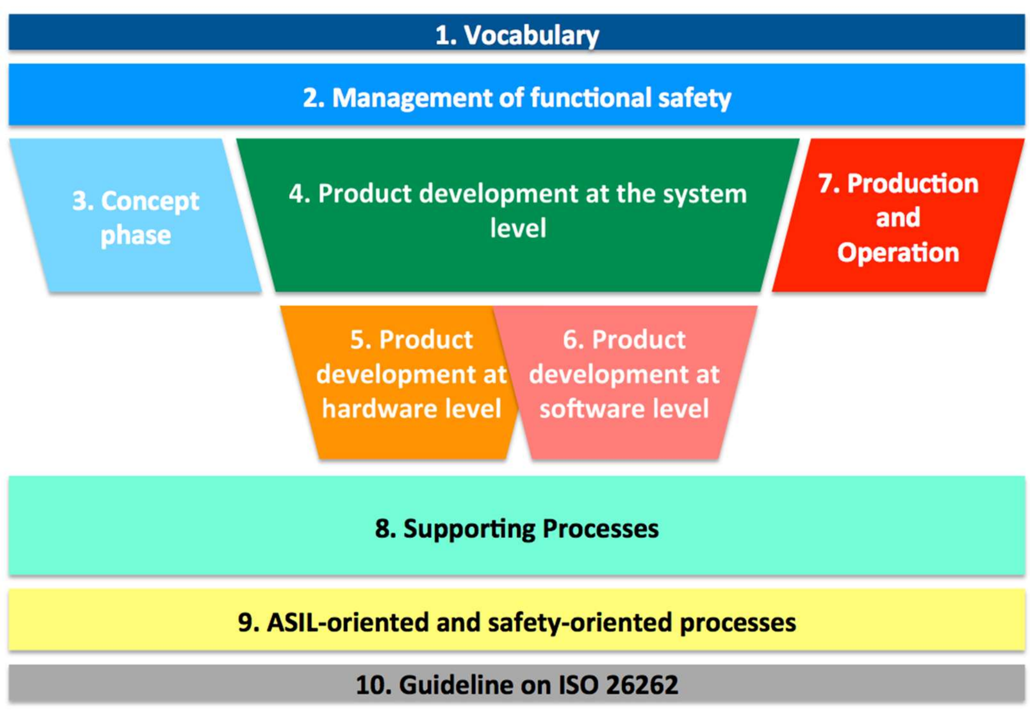


Figure 2 ISO26262 Block Diagram

2.2 Importance of ISO 26262 in NVMPC Verification

The Non-Volatile Memory Platform Controller (NVMPC) is an important part of automotive systems that is in charge of managing non-volatile memory. The NVMPC is in charge of storing and managing important data like firmware, configuration settings, and diagnostic information. This makes it very important to make sure that the NVMPC is safe to use. If something goes wrong with the NVMPC, there's a chance that data could get corrupted or lost, which could be very dangerous.

2.2.1 Why ISO 26262 is Required for NVMPC:

Data Integrity and Reliability:

Requirement: The NVMPC is in charge of managing non-volatile memory, which can save data even when the power is off. Because of this, it is important to make sure that the data is correct and reliable at every stage of ISO 262 in order to follow the rules: The standard says that strict verification procedures must be used to make sure the data is correct. These include systems that can find and fix mistakes. This helps keep data safe and sound, which is very important for keeping car systems safe and reliable.

Fault Tolerance and Error Handling:

Requirement: The NVMC needs to be able to handle faults well and keep functional safety even when mistakes happen. The standard says that fault injection testing and formal verification must be done to make sure that the NMMPC can find and fix mistakes. This is in line with ISO 26262. This also makes sure that the controller ISO can fix mistakes without putting the system's safety at risk.

Compliance with ASIL Levels:

Keeping up with ASIL Levels: To make sure that the NVMPC is as safe as possible, it is very likely that it will have to meet higher ASIL standards, like ASIL D. The importance of the data it handles will determine this. To follow the rules: The standard ISO 26262 tells you how to meet the requirements for different ASIL levels, including specific ways to check and confirm that you are following the rules. Also, there are specific steps that are taken to check and confirm. Because of this, the NVMPC is guaranteed to meet the necessary level of safety integrity for the jobs it was made to do.

2.3 Conclusion

ISO 26262 is an important standard that makes sure car systems are safe to use. One of the most important parts of this standard is the Non-Volatile Memory Platform Controller (NVMPC). Standard IS0 26262 has all the rules for risk assessment, verification, validation, and documentation that need to be followed to make sure the NVMPC meets very high safety standards. Because of this, there are fewer chances of problems that could cause dangerous situations, which keeps car systems safe, reliable, and honest. To keep high standards of functional safety in the automotive industry, it is very important to follow ISO 26262 very closely.

Chapter 3: Literature Review

3.1 Literature section

M. J. Prajwala [1] looked into the problems with verifying NAND flash memory controllers, which are now important parts of consumer electronics and business storage solutions. The paper went into great detail about how NAND controllers have changed over time to handle wear leveling, error correction, and garbage collection. The authors used System Verilog-based verification methods to show how to effectively check that a controller works in a variety of operational situations. They stressed how important it is to have strong verification to make sure that data is safe and that controllers work reliably as NAND flash technology gets bigger. The study also talked about how flash memory systems are getting more complicated and how we need verification strategies that can adapt to different workloads. Lee et al. gave us information on how to design testbenches, coverage metrics, and fault injection methods that are specific to NAND controller verification. Their work helps make flash storage solutions that are more reliable and efficient.

K.S.Pooja [2] used a limited random-based testbench made with System Verilog and UVM to do a detailed study on how to verify interconnection IP for automotive applications. Their method combined assertions, functional coverage, and code coverage to make sure that automotive IPs were fully tested. The paper talked about how automotive systems are getting more complicated and how important it is to have reliable verification methods to meet strict safety standards. By using UVM, the authors showed that testbenches could be made more modular and reusable, which is important for keeping up with changing automotive standards. It was shown that the method made finding and verifying faults more efficient. This work is important because it helps solve problems that come up in safety-critical automotive environments where IP interconnectivity is very important. The study also talked about how to put their verification strategy into action and how to measure coverage to show that it works. Overall, it helps move forward verification methods that are specific to automotive IPs, making sure that systems are reliable and follow the rules.

Y. Yun [3] looked at the problems with traditional System Verilog testbenches, especially how they can't be reused or scaled up for complex SoC verification situations. To get around these problems, the authors suggested using the Universal Verification Methodology (UVM), which provides a standardized framework that encourages the reuse and consistency of verification components. Their study went into detail about how UVM testbenches are built and how they help with better test management and automation. They talked about how UVM can improve the quality of verification, speed up development, and make it easier to maintain. The paper also talked about real-world problems that come up when using UVM and offered solutions to make the switch from old methods easier.

A. Ismail [4] looked into how automotive electrical and electronic (E/E) systems are getting more complicated, especially when it comes to safety-critical applications that follow ISO 26262 standards. The paper talked about how different advanced car features work together and how this makes it harder to make sure the car is safe to drive. Some of the main topics covered are system-level errors, random hardware faults, and how they affect the reliability of the whole system. The authors talked about how E/E components are becoming more tightly integrated and interconnected, which makes it harder to verify and manage faults. They looked at current methods for finding and fixing faults and found areas that need to be improved in order to keep up with changing safety standards. The study also showed how important it is to use a wide range of verification methods, such as formal methods, simulation, and fault injection. By giving a detailed look at these problems, it helped people understand how hard it is to check the safety of modern car systems.

R. Madan [5] focused on practical ways to add self-checking features to UVM-based testbenches in order to make verification easier and less time-consuming to debug. The paper talked about a number of ways to do self-checking, such as reference model checkers and scoreboard-based verification. This gave engineers the freedom to choose the methods that worked best for their projects. The authors talked about the pros and cons of different approaches in terms of how easy they are to scale, how complicated they are, and how easy they are to put into action. They showed how adding self-checking features makes it easier to find mistakes and speeds up the process of verifying them. The study also talked about the benefits of vertical reuse in UVM settings and suggested ways to make it easier to develop checkers. Singh et al. gave us useful information on how to make UVM testbenches more robust and efficient in complex verification projects by giving us detailed instructions and examples.

Zhang [6] suggested a useful and efficient SoC verification flow that makes use of IP test cases and testbenches made with UVM. The paper talked about some of the most common problems with SoC verification, like getting different IP teams to work together and dealing with SoC-IP interface problems. Their method focused on modular verification and incremental integration, which made it easier to find and fix bugs more quickly. Chen et al. showed that their method worked by using it on a real-world SoC project. They found that the verification cycle time was cut in half and the number of engineers needed was greatly reduced. The study also talked about ways to automate and manage verification that help with scalable SoC verification. This work gives us a useful framework for making verification more productive and of higher quality in complicated SoC designs.

H. Zhaohui [7] using the Universal Verification Methodology (UVM), we made a full set of tests for checking SoC interconnects. The paper went into great detail about how to design reusable test sequences and scenarios that deal with the complexity of today's SoC interconnect architectures. The authors stressed how important it is to thoroughly check interconnects to make sure that data is correct, timing is correct, and protocols are followed. Their test suite used coverage-driven

verification methods and fault injection to find failures that only happen in certain situations. Rao et al. also talked about problems that come up when trying to make interconnect verification components work on more than one SoC project. The study showed that verification coverage and efficiency were better, which led to more reliable SoC communication infrastructures. Their work is a useful guide for verification engineers who have to check complicated interconnects.

S. A. Saji [8] gave a detailed overview of how to design fault-tolerant systems in the automotive industry. The paper talked about architectural strategies, such as redundancy, error detection, and correction mechanisms, that can make systems more reliable at both the integrated circuit and full-system levels. The writers stressed how important fault tolerance is for meeting strict automotive safety standards like ISO 26262. They looked at the most advanced ways to manage faults and how they could be used in different parts of a car. The study also looked at the problems of adding fault tolerance without spending too much money or using too much power. Khan et al. stressed how important it is to use integrated design and verification methods to make sure that fault tolerance is strong. Their thorough review helps designers choose the right fault-tolerant architectures for use in cars.

A. Manzone [9] reviewed the current status and challenges in SoC verification for the embedded systems market, focusing on IP reuse, pre-verified platforms, and hardware-software co-development. The paper highlighted the increasing complexity of SoCs and the need for scalable verification methodologies. The authors discussed platform-based design approaches that enable modular verification and incremental integration. They emphasized the role of verification reuse in reducing development time and improving quality. Gupta et al. also explored challenges related to verification tool limitations and the integration of heterogeneous IPs. Their analysis underscored the importance of developing flexible verification frameworks that can adapt to evolving system requirements. The study provides insights into future directions for SoC verification in embedded systems.

Choi [11] gives a UVM-based way to check flash memory with Error Correction Codes (ECC). It uses a SystemVerilog testbench and constrained-random testing to make sure that all functions are tested thoroughly. The testbench has drivers and monitors that can be used again and again. It focuses on ECC to fix problems with data integrity, such as bit flips. The method puts ECC algorithms through a lot of stress tests and uses coverage metrics to check their reliability. The results of the simulation show that UVM is good for scalable flash memory verification because it can find and fix more errors.

3.2 Research gaps

i. **Automation Needs Improvement** : Current automation tools can't always handle really complex systems well and need to become smarter and more flexible.

ii. **Better Integration Needed** : Combining formal methods with simulation is helpful, but making them work smoothly together in one process is still tricky.

iii. **More Accurate Fault Sorting** : We need better ways to automatically tell which faults are critical and which are safe to focus verification efforts wisely.

iv. **Scaling to Big Systems is Hard** : Verifying very large and mixed systems remains tough, especially when many different IP blocks are involved.

v. **New Tech Needs New Methods** : Verification approaches for new technologies like advanced memory or AI chips are still not fully developed.

vi. **Hardware-Software Teamwork** : There's a need for better tools and methods to help hardware and software teams work together during verification, especially for safety-critical projects.

Chapter 4: Problem Statement

One of the main goals of this study is to come up with and put into action a full verification plan for the Non-Volatile Memory Platform Controller (NVMPC) IP. 1S0 26262 has very high standards for safety and dependability, and this plan should be able to meet them. To make sure that the NVMPC is safe to use in cars, we need to fix the problems and flaws with the current verification methods and make the whole verification process better.

- **Develop a Comprehensive Verification Environment**
  Objective: Make a verification environment that includes everything. The goal of this project is to make a verification environment for the NVMPC IP that is both reliable and scalable.

- **Enhance Coverage Metrics and Analysis**
  Objective: Better performance metrics and coverage analysis. The goal is still to get full coverage of all the NVMPC's working features.

- **Create Reusable Verification Components**
  Objective: Make components for verification that can be used again. The goal is to make verification sections that can be used more than once. The goal is to make verification parts that are not only strong but also flexible and can be used more than once.

- **Automate Compliance Monitoring and Updates**
  Objective: The implementation of automatic monitoring and real-time updates is going to be carried out in order to guarantee that ISO 26262 is always applied.

- **Perform Comprehensive Regression Testing**
  Objective: To make sure that ISO 26262 is always followed, automatic monitoring and real-time updates will be put in place.

- **Document and Disseminate Knowledge**
  Objective: The purpose of this program is to document the verification process and then make that knowledge available to the entire technical community.

- **Ensure Compliance with ASIL Levels**
  Objective: Guarantee that the ASIL values are adhered to at all times.
  In order to accomplish this, it is necessary to make certain that the NVMPC satisfies the Automotive Safety Integrity Level (ASIL) standards for the application that it was developed for.

Chapter 5: IP Verification Methodology

The most important part of the method is to understand the design. Until the objective is accomplished, it illustrates the entire proof process, beginning with the planning stage and ending with the results. A coverage grid, a verification pan, and a verification plan back annotation are all included in it. The first step is to include all of the design features and regulations in a file that is referred to as BG, which is an abbreviation that stands for "block guide."

The knowledge in question originates from the design engineer, and a strategy is developed on the basis of the BG verification information. There are multiple teams that review the feature collection before it is sent to the verification team for the final approval. This is a very hierarchical position. The verification plan, which we refer to as vplan, is the following plan in the sequence. It is a piece of paper that contains a list of all of the test cases that will be administered in order to validate the effectiveness of the strategy.

The tools generate a coverage report, modelling findings, and waveforms whenever we conduct regression or simulations. These are all examples of what we may expect. Cadence's vManager tools are being utilised in this particular instance. All of the logs and results are included in here. Through the use of this tool, coverage charts and test cases are evaluated and either passed or failed. When the coverage arrives, it examines the code coverage as well as the functional coverage to determine whether or not they are accurate. The verification team will not shut it until all of the features have been tested, the test cases have been cleaned up, and the coverage reports demonstrate that all of the features have been tested.

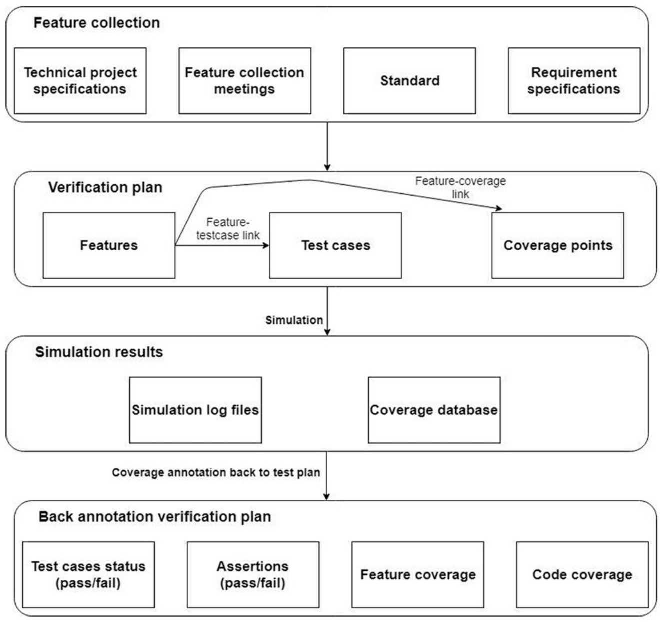


Figure 3 Verification management flow

5.1 Verification Methodology

It is necessary to make improvements to verification methods since the design of chips is becoming more difficult and the market is changing. The use of an SV-based methodology, a UVM-based method, or an assertion-based proof are just some of the many solutions that are available for accomplishing this task.

5.2 SV Based Methodology System

Verilog is responsible for the introduction of the principle of OOPS, which simplifies the verification process. Because of this, the verification environment is able to undergo changes and make use of randomisation techniques and classes. A test bench that is based on system verilog is comprised of the following components: a generator, a driver, a monitor, an agent, a scoreboard, an environment, a test program, an interface, and other top modules. All of these are responsible for a variety of functions, including the creation of packets and the transmission of those packets to the DUT.[25]

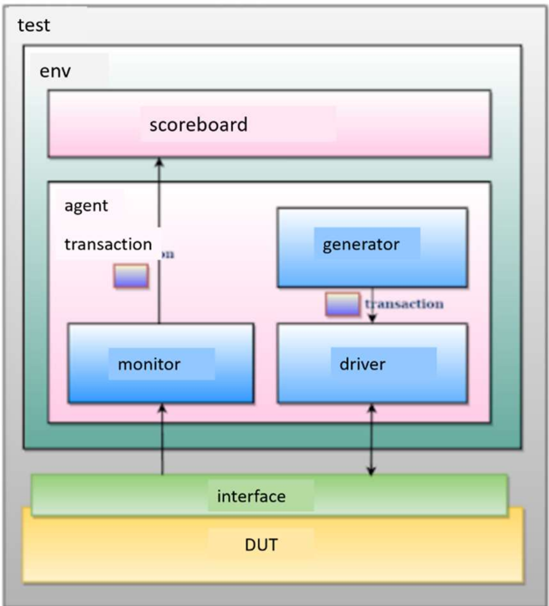


Figure 4 SV Testbench

5.3 UVM Based Methodology

In terms of design, the Universal Verification Methodology (UVM) is comparable to the SV, despite the fact that UVM possesses a greater number of features. Through the utilisation of the UVM-based technique, it is possible to repeatedly utilise testbench settings that were created by utilising particular classes. To establish a reliable verification environment, SV was instrumental in the development of this fundamental industry standard. When performing various operations, UVM makes use of pre-defined classes. An example of this would be sequences that require a UVM_Driver, which is an extension of the sequence class, in order to be driven. A factory functionality is available in UVM, which enables you to register and obtain classes and objects without requiring any other testbench

files to be modified. In order to maintain a consistent flow and to keep track of all of the numerous actions that are taking place during simulation, UVM makes use of the phasing mechanism [26].

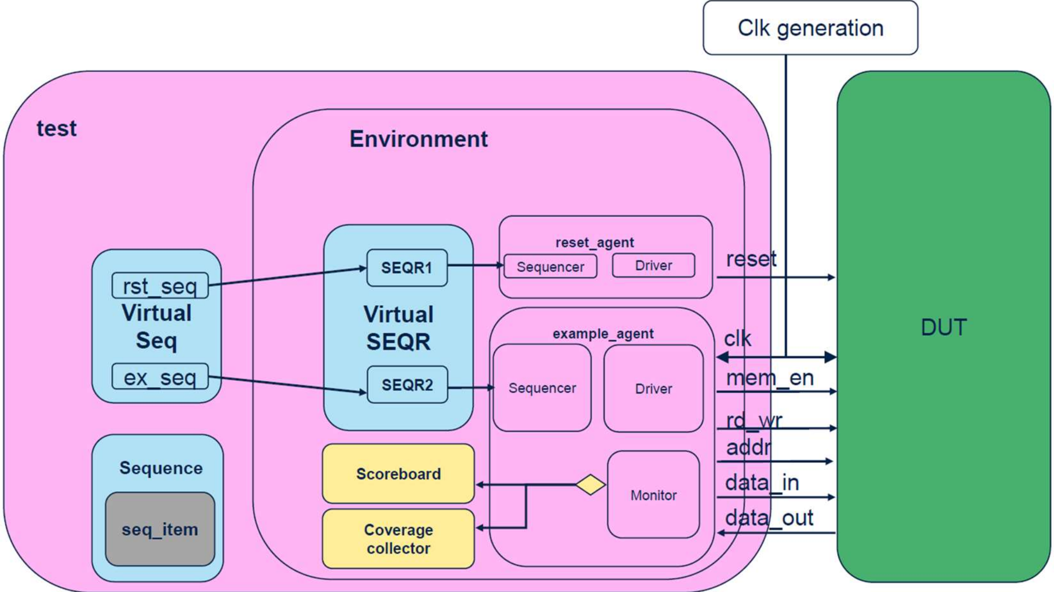


Figure 5 UVM Testbench

5.4 Assertion Based Methodology

It is checked by assertions to see if the plan meets the requirements. If the qualities are wrong, then the statements are wrong too. Statements come in a lot of different forms.

1. Declarements on the Spot: This tool is used to evaluate different situations right now. Example: As an example, always_comb start x_eq_y: if x == y, then ("x not equal y"); end

2. Statements Made at the Same Time: It looks at how things happen in a certain order over a lot of short clock cycles. It's called "Property" when that happens. Example: x_eq_y: assert property ((@posedge clk) z |=> (x == y));

Chapter 6: Work done

The Universal Verification Methodology (UVM) environment for testing the NVMPC IP is structured to ensure comprehensive verification of the Design Under Test (DUT).

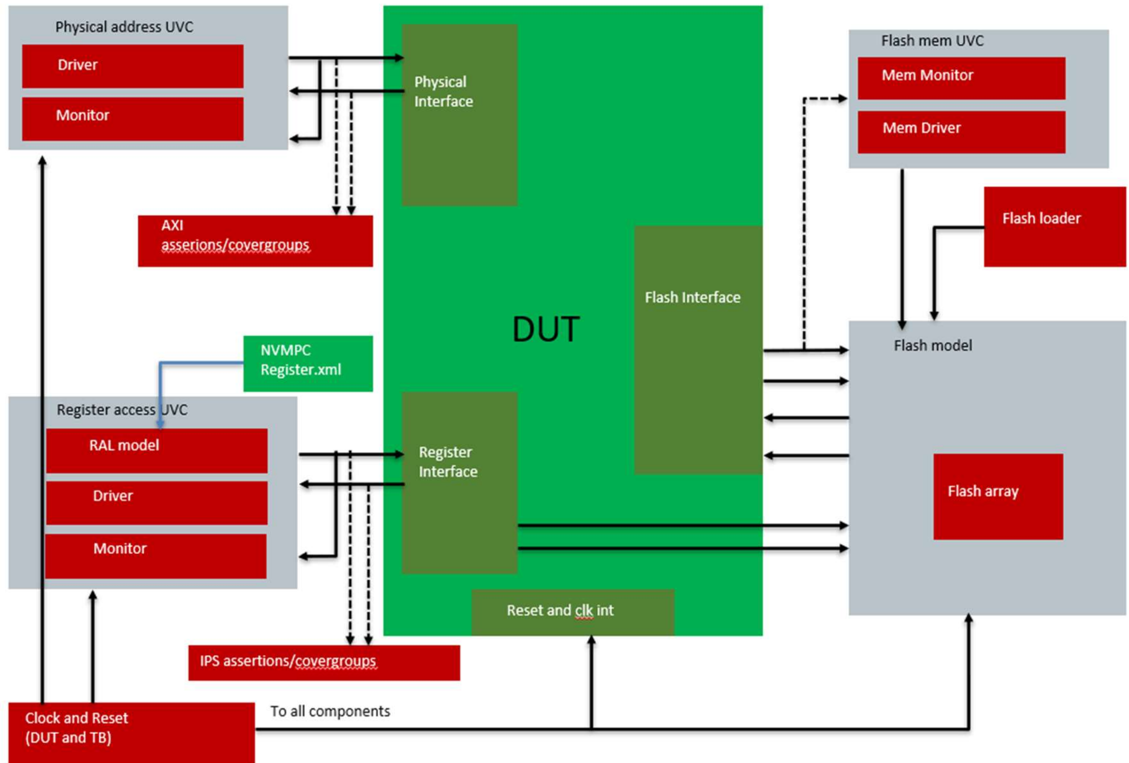


Figure 6 NVMPC UVM Environment

6.1 Blocks and components:

1. **Physical address UVC (Universal Verification Component) :**

   Driver: The driver is in charge of doing AXI transactions and sending them to the DUT's AXI interface.

   Monitor: Watches the AXI4 transactions happening on the bus and makes sure they follow the rules that have been set.

   AXI Assertions/Covergroups: It is in charge of making sure that the AXI protocol is being used correctly and collecting data on coverage.

2. **Register access UVC :**

   RAL Model: RAL Model stands for Register Abstraction Layer model. It gives you a high-level way to access the DUT's registers.

   Driver: Sends transactions to IPS interface of DUT.

   Monitor: Checks the transactions on the IPS interface to make sure they are correct.

IPS Assertions/Covergroups: Makes sure that the IPS protocol is set up correctly and gathers data on coverage.

3. **Flash Memory UVC :**
   Mem Driver: Sends transactions to the flash memory interface.
   Mem Monitor: watches the transactions on the flash memory interface and makes sure they follow the rules.
   Flash Loader: Loads data into flash model so it can be tested.

4. **Flash Model :**
   Flash Array: This program simulates the flash memory array, giving you a real-world setting to test the DUT's flash interface.

5. **DUT (Design Under Test) :**
   AXI Interface: The Interface for AXI transactions.
   Register access Interface: This is the interface for transactions that access registers.
   Flash Interface: This is the interface to talk to the flash memory.

6. **Clock and Reset :** Sends clock and reset signals to all parts of the testbench and DUT.

6.2 Flow of AXI signals

There are a few main parts that make up the flow of signals in the verification environment.

The Transaction Driver starts transactions and sends them to the Device Under Test (DUT) interface. The Transaction Monitor keeps an eye on these transactions to make sure they follow the rules.

Tools for Transaction Assertions and Coverage gather coverage data and check that the implementation is correct. The Register Driver sends transactions that need to access the register to the register interface. The Register Assertions/Coverage checks that the protocol is correct and collects coverage data. The Register Monitor checks that the transactions are correct. The Register Abstraction Model is a high-level interface that makes it easier to get to and look at the DUT's registers.

The Memory Driver sends data to the memory interface for memory transactions. The Memory Monitor checks these transactions to make sure they follow the rules. The Memory Loader then loads data into the Memory Model to make it look like the memory array.

The Clock and Reset block sends the signals that all of the parts need to work together. This makes sure that the DUT and the testbench are always on the same page.

There are many Verification Components (UVCs) in the UVM testing environment for the IP block. These UVCs work together to let the Device Under Test (DUT) send, receive, and check transactions on its interfaces.

Each UVC is in charge of transactions that happen on a certain interface. This means making things happen and watching how people respond. We can look at and access registers with a Register Abstraction Model. Assertions and coverage techniques, on the other hand, get coverage data and make sure that rules are followed. The Clock and Reset block sends clock and reset signals to all of the components at the same time. This exact setup makes sure that the IP block is fully tested, showing that it works as it should and follows the rules.[27]

6.3 Testbench structure

The following diagram illustrates the test bench directory structure that we at STMicroelectronics employ for all IP verification applications. In addition to that, this includes the subdirectories that are shown below, which are used to organise files according to how they are utilised in the test bench. This is a rudimentary outline of the structure that will be addressed, which may be found below.

STA1_NVMPC_IPS_TB

- ├── README : Contains vplan and verification report
- ├── docs : Contains all design files of NVMPC
- ├── dut : Contains all design files of NVMPC
- ├── run : Simulation run directory with all simulation scripts
- ├── sv : Contains System Verilog files for all UVC and virtual sequencer library
- ├── tb : Contains testbench
- ├── test : Test sequence files
- ├── vmanager : vManager directory with regression vsif files and vManager setup

6.4 Verification planning

A vPlan is a clear plan that explains how to check if an IP block works correctly. It shows what needs to be tested, how to test it, and what tools or resources are needed. It also gives you a schedule and goals for testing. A good vPlan makes sure that all parts of the IP are tested properly and that any problems are found early.
Tool used: Verisium Manager - Cadence

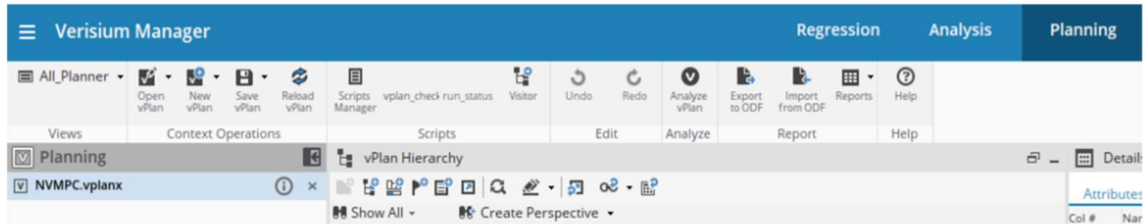


Figure 7 Verisium manager tool

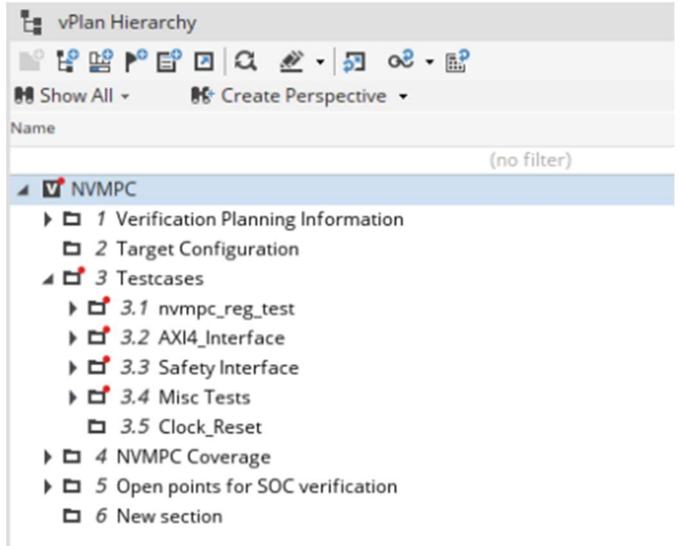


Figure 8 vPlan structure

6.5 UVM Components

A UVM is made up of parts that are linked together to make a complex test bench for checking integrated circuits. These parts are arranged in a hierarchy and are meant to handle different parts of the verification process, like generating stimuli and showing signals. The main parts of UVM are "uvm_test," which sets up the verification environment, and "uvm_env," which describes the whole test bench architecture, "uvm_agent," which is in charge of certain protocol tasks, and "uvm_driver" and "uvm_monitor," which make test stimuli and watch the DUT's outputs. This framework supports reusability and scalability, which is why UVM is a popular choice for thorough and efficient verification processes.[27]

6.5.1 *UVM Sequence and sequence item*

The sequences are implemented using the UVM sequence class. To stimulate the DUT, sequence items are designed with data fields that model the test scenarios. Custom sequence items derived from the UVM sequence item class are useful for leveraging inherited functions which streamline development and debugging processes. These data fields are typically randomized with constraints to ensure realistic test conditions, and to the randomized nature of the test bench environment.

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

  function new (string name = "");
    super.new(name);
  endfunction

  `uvm_object_utils_begin( my_seq_item )
    `uvm_field_int( data, UVM_ALL_ON )
    `uvm_field_int( addr, UVM_ALL_ON )
  `uvm_object_utils_end

endclass : my_seq_item
```

Figure 9 UVM Sequence item

6.5.2 *UVM Sequencer*

The sequencer is created by extending the UVM sequencer class with the sequence item type as a parameter. It's a sophisticated stimulus generator that generates and returns data items in response to the driver's commands. By altering the randomization weights, a sequencer follows the present condition of the DUT and generates more valuable data items.

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

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

endclass : axiSequencer
```

Figure 10 UVM Sequencer

6.5.3 *UVM Driver*

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

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

typedef axiDriver #(PARAM)          axiDriver_t;
typedef axiTransaction #(PARAM)     axiTransaction_t;
typedef virtual interface axi_if #(PARAM) axi_if_t;
`uvm_component_param_utils(axiDriver_t)

bit reset_done;
bit packet_received;
axi_if_t axi_if_h;

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

Figure 11 UVM Driver

6.5.4 *UVM Monitor*

This component operates independently to monitor all interactions between the DUT and the test bench. It captures signals and forwards them to other modules like scoreboards for analysis. If communication deviates from a set of standards or protocols, the monitor flags errors. It translates signal-level data into sequence item format and interfaces with specific signals from each UVC, transmitting data to the scoreboard through an analysis port.

```
class AXIMonitor #(type PARAM=int) extends uvm_monitor;
  typedef AXIMonitor          #(PARAM) AXIMonitor_t;
  typedef AXITransaction      #(PARAM) AXITransaction_t;
  typedef virtual interface axi_if #(PARAM) axi_if_t;
  `uvm_component_param_utils(AXIMonitor_t)
```

Figure 12 UVM Monitor

6.5.5 *UVM Agent*

User-defined protocols in the verification environment use the UVM agent, which comes from the UVM component. It brings together important parts like drivers, sequencers, and monitors. There are many agents in one UVC that work with different protocols in SoC designs. The agent connects through stages to create and link components that are different from the run phase. Depending on their roles and needs, agents can work actively or passively..

```
class axiAgent #(type PARAM=int) extends uvm_agent;

  uvm_active_passive_enum is_active = UVM_ACTIVE;
  //`uvm_component_utils(AuthAgent)

  typedef axiAgent #(PARAM) axiAgent_t;
  `uvm_component_param_utils_begin(axiAgent_t)
    `uvm_field_enum(uvm_active_passive_enum, is_active, UVM_ALL_ON);
  `uvm_component_utils_end

  typedef axiSequencer #(PARAM) axiSequencer_t;
  typedef axiDriver #(PARAM) axiDriver_t;
  typedef axiMonitor #(PARAM) axiMonitor_t;
```

Figure 13 UVM Agent

6.5.6 *UVM Scoreboard*

A verification scoreboard keeps an eye on the flow of data into and out of the DUT to make sure it works correctly. It looks at the inputs that go into the test environment and the outputs that the DUT makes. The scoreboard collects data from multiple sources, processes it independently, and compares the actual outcomes with the expected results to validate the DUT's performance.

```
class axi_score_board#(int FLASH_ADD_WIDTH = 29) extends uvm_scoreboard;
  `uvm_component_param_utils(axi_score_board#(FLASH_ADD_WIDTH ))

   uvm_analysis_export #(denaliCdn_axiTransaction) trans_in;
   uvm_analysis_export #(flash_mem_seq_item#(FLASH_ADD_WIDTH )) trans_out;
   uvm_analysis_export #(denaliCdn_axiTransaction) resp_in;
   uvm_analysis_export #(flash_mem_error_seq_item#(FLASH_ADD_WIDTH )) resp_out;

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

endclass
```

Figure 14 UVM Scoreboard

6.5.7 *UVM Environment*

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

```
class st_axi4_Env#(int FLASH_CONTROLLER = 0) extends uvm_env;

  `uvm_component_param_utils(st_axi4_Env#(FLASH_CONTROLLER))
  // ****************************************************************
  // The environment instantiates Master and Slave components
  // ****************************************************************
  st_axi4_Agent activeMaster;
  st_axi4_Agent passiveSlave;
 //for write
  st_axi4_Agent activeMaster_write;
  st_axi4_Agent passiveSlave_write;
//  st_axi4_Agent activeSlave;
  configuration_class config_class;
```

Figure 15 UVM Environment

6.5.8 *UVM Test*

Test generates the test bench scenarios by carrying out the sequencer to define which sequence will generate when a particular test case is run. This allows the user to select data items to transmit to the DUT based on their requirements. The global run test assignment must be specified in the test block. Test is written as per the basic specifications given by the design team and verification plans made by the verification teams. They are written to check the specific scenarios of particular IP/SoC where SoC handshaking process of C and SV is done to make a verification of functionalization of the test.

6.5.9 *UVM Top*

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

```
`ifndef CDN_AXI_UVM_USER_TOP
`define CDN_AXI_UVM_USER_TOP

`include "cdnAxiUvmDriver.sv"
`include "st_axi4_Types.sv"
`include "st_axi4_Instance.sv"
`include "st_axi4_MasterDriver.sv"
`include "st_axi4_Monitor.sv"
`include "st_axi4_Sequencer.sv"
`include "st_axi4_Agent.sv"
`include "st_axi4_Env.sv"
`include "st_axi4_VirtualSequencer.sv"

`endif // CDN_AXI_UVM_USER_TOP
```

Figure 16 UVM top

6.6 UVM Phases

UVM (Universal Verification Methodology) divides the testbench operation into several **phases** to organize the verification process. Each phase has a specific role. Major phases are:

6.6.1 *Build phase*

The **build_phase** is where all the components of the testbench are created and initialized. It sets up everything needed before the test can start running.

```
function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      exp          = new("exp", this);
      out          = new("out", this);
      expfifo      = new("expfifo", this);
      outfifo      = new("outfifo", this);
      end_of_trans = new("end_of_trans",this);
endfunction : build_phase
```

Figure 17 UVM Build phase

- super.build_phase(phase); calls the parent class's build_phase to ensure any inherited setup is done.
- Each object is given a name (like "exp") and linked to the current component (this).
- These objects could be FIFOs, events, or other components needed in the testbench.

6.6.2 *Connect phase*

The **connect_phase** is where different components in the testbench are linked together. This phase ensures that data and events flow properly between components by connecting their ports and

```
function void connect_phase(uvm_phase phase);
      super.connect_phase(phase);
      exp.connect(expfifo.analysis_export);
      out.connect(outfifo.analysis_export);
endfunction : connect_phase
```

Figure 18 UVM connect phase

exports.

- super.connect_phase(phase); calls the parent class's connect_phase to perform any inherited connections.
- The method connects exp to the analysis_export of expfifo, allowing exp to receive data or events from expfifo.
- Similarly, out is connected to the analysis_export of outfifo.
- This setup enables communication between these components, so data can be passed and monitored during simulation.

6.6.3 *Run phase*

The **run_phase** is where the actual testing happens. In this phase, stimulus is sent to the design, responses are collected, and checking or monitoring occurs. It usually runs concurrently with other activities and can continue until the test is complete.

```
task run_phase(uvm_phase phase);
      int que_size;
      flash_mem_seq_item#(FLASH_ADD_WIDTH ) exp_tr, out_tr;
      denaliCdn_axiTransaction resp_exp_tr;
      flash_mem_seq_item#(FLASH_ADD_WIDTH ) temp;
      flash_mem_error_seq_item#(FLASH_ADD_WIDTH ) resp_out_tr;
      if(uvm_config_db#(int)::get(this,"","disable_checking",disable_checking))
          $display("disable_checking = %h",disable_checking);
      if(!disable_checking) begin
        forever begin
          fork
```

Figure 19 UVM Run phase

6.6.4 *Extract phase*

The **extract_phase** is used to collect and process data after the main test run. It often checks for any leftover or unexpected transactions in FIFOs or buffers that should have been handled during the test. This helps identify issues like missed responses or incomplete processing.

```
// Function: extract_phase - checks leftover jelly beans in the FIFOs
//----------------------------------------------------------------------
virtual function void extract_phase( uvm_phase phase );
   denaliCdn_axiTransaction resp_exp_tr;
   flash_mem_error_seq_item#(FLASH_ADD_WIDTH) resp_out_tr;

   super.extract_phase( phase );
   if(!disable_error_checking) begin
      if ( read_transactions.try_get(resp_exp_tr) ) begin
         `uvm_error( "resp_in_fifo",
                     { "found a leftover  ", resp_exp_tr.convert2string() } )
      end
      if ( read_beat_transactions.try_get(resp_exp_tr) ) begin
         `uvm_error( "resp_in_fifo",
                     { "found a leftover  ", resp_exp_tr.convert2string() } )
```

Figure 20 UVM Extract phase

6.7 Output waveform

The waveform diagram offers a comprehensive perspective of the protocol signal flow, beginning with the master core (a_master0) and ending with the slave NVMPC IP (p_slave). The waveform diagram shows this view. This study has shown that the protocol is working correctly, which means that communication will be reliable and data will be safe throughout the whole system.

Figure 21 Waveform protocol signal flow

6.8 Observations:

The waveform shows that an AXI read transaction went from the master (a_master) to the slave (top_read) and worked. At every step, timing and data integrity are kept up, and the data transfer and signal propagation are always correct. This study shows that the AXI protocol works right in this system, so the core, DUT, and child parts can all talk to each other without any problems.

1. **Initiation of Read Transaction**: This step starts a read transaction, which sets up the needed information and sends it throughout the system.

2. **Propagation of Address and Control Signals**: The master sends the address and control information to the slave in a way that makes sure the slave gets the right information.

3. **Read Data Validity**: The read data is only valid if it is available in a sequence across all modules. This helps keep the data consistent.

4. **Read Response**: The read response works perfectly in all modules, with no problems, and it matches the original transaction ID.

5. **Timing Synchronization**: The timing of signals is well-coordinated across all modules, which ensures that data transfer is carried out in an effective manner

6. **Data Integrity and Reliability**: The communication protocol's dependability and robustness are validated by the fact that information is consistently propagated throughout all modules through the communication protocol.

6.9 Testcases

Below are some excerpts from my testcases in the project

6.9.1 Basic test sequence

```
// ****************************************************************************
// Class : basicTest
// Desc. : send several bursts
// ****************************************************************************
class basicTest extends nvmpc_base_test;
  `uvm_component_utils(basicTest)

  function new(string name = "basicTest",uvm_component parent);
    super.new(name,parent);
  endfunction : new

  virtual function void build_phase(uvm_phase phase);
   uvm_config_db#(int)::set(this,"axi_sve","initialize_random_data_in_flash",0);
   uvm_config_db#(uvm_object_wrapper)::set(this, "axi_sve.vs.run_phase", "default_sequence",
AXIReadSeq_vs::type_id::get());
   uvm_top.set_config_int("*", "recording_detail" , UVM_FULL);
   super.build_phase(phase);
   config_class.change_system_address(INDIVIDUAL_BLOCK);
  endfunction : build_phase

endclass : basicTest
```

Figure 22 Basic test

6.9.2 Reset value check test

```
// ****************************************************************************
// Class : nvmpc_reset_value_vs
// Desc. : nvmpc_reset_value_vs virtual sequence to check reset value of nvmpc registers
// ****************************************************************************
class nvmpc_reset_value_vs extends nvmpc_VirtualSequence;
  `uvm_object_utils(nvmpc_reset_value_vs)
    function new(string name="nvmpc_reset_value_vs");
    super.new(name);
  endfunction
```

Figure 23 Reset value check test

6.9.3 Reg access test

```
// Sequence: Perform 32 bit write and read compare check to all register of DUT
//-------------------------------------------------------------
class nvmpc_reg_32bit_access_vs extends nvmpc_VirtualSequence;
 `uvm_object_utils(nvmpc_reg_32bit_access_vs)
 function new(string name="nvmpc_reg_32bit_access_vs");
  super.new(name);
 endfunction

 uvm_reg_hw_reset_seq   uvm_reg_hw_reset_seq_h;
 write_read_compare_seq write_read_compare_seq_h;

 virtual task body();

 uvm_report_info(get_full_name(),"Write hFFFFFFFF in all registers ", UVM_MEDIUM);
 write_read_compare_seq_h             = write_read_compare_seq :: type_id::create("write_read_compare_seq");
 write_read_compare_seq_h.model       = nvmpc_reg_model;
 write_read_compare_seq_h.write_value = 32'hFFFFFFFF;
 write_read_compare_seq_h.start(p_sequencer.ipsbus_seqr_h);
```

Figure 24 Register access test

6.9.4 Endian check test

```
// Sequence: Perform endianess check to all register of DUT

class nvmpc_reg_endian_vs extends nvmpc_VirtualSequence;
 `uvm_object_utils(nvmpc_reg_endian_vs)
 function new(string name="nvmpc_reg_endian_vs");
  super.new(name);
 endfunction

 reg_32bit_endian_seq reg_32bit_endian_seq_h;

 virtual task body();

 uvm_report_info(get_full_name(),"starting sequence reg_32bit_endian_seq", UVM_MEDIUM);
 reg_32bit_endian_seq_h       = reg_32bit_endian_seq :: type_id::create("reg_32bit_endian_seq");
 reg_32bit_endian_seq_h.model = nvmpc_reg_model;
 reg_32bit_endian_seq_h.write_value = 32'h12345678;
 reg_32bit_endian_seq_h.start(p_sequencer.ipsbus_seqr_h);
 endtask :body

endclass : nvmpc_reg_endian_vs
```

Figure 25 Endian check test

6.10 Simulation results:

During the beginning stages of the simulation, it is essential to check the log file for uvm_error and uvm_info. Following that, the waveform is used to verify the intended behaviour, and the passing and failing status of the testcase are recorded. An application called Cadence SimVision Xcelium is being utilised by us for the purpose of waveform analysis and simulation.

```
virtual function void report();
        int count;
        count = server.get_severity_count(UVM_ERROR);
        if (count > 0)
        begin
         $display("INFO END_SIM:                                                   ; T=%0d", $stime);
         $display("INFO END_SIM:           FFFFFF     AA       II   LL             ; T=%0d", $stime);
         $display("INFO END_SIM:           FF        AAAA      II   LL             ; T=%0d", $stime);
         $display("INFO END_SIM:           FFFFF    AA  AA     II   LL             ; T=%0d", $stime);
         $display("INFO END_SIM:           FF      AAAAAAAA    II   LL             ; T=%0d", $stime);
         $display("INFO END_SIM:           FF     AA      AA   II   LL             ; T=%0d", $stime);
         $display("INFO END_SIM:           FF    AA        AA  II   LLLLLL         ; T=%0d", $stime);
         $display("INFO END_SIM:                                                   ; T=%0d", $stime);
         $display("INFO END_SIM: Simulation completed with Errors!!; ERRORS_COUNT=%0d", count, $stime);
        end
        else if(count == 0)
        begin
         $display("INFO END_SIM:                                                   ; T=%0d", $stime);
         $display("INFO END_SIM:           PPPPP      AA        SSSS    SSSS       ; T=%0d", $stime);
         $display("INFO END_SIM:           PP  PP    AAAA      SS  SS  SS  SS      ; T=%0d", $stime);
         $display("INFO END_SIM:           PPPPP    AA  AA      SS      SS         ; T=%0d", $stime);
         $display("INFO END_SIM:           PP      AAAAAAAA       SS      SS       ; T=%0d", $stime);
         $display("INFO END_SIM:           PP     AA      AA   SS  SS  SS  SS      ; T=%0d", $stime);
         $display("INFO END_SIM:           PP    AA        AA   SSSS    SSSS       ; T=%0d", $stime);
         $display("INFO END_SIM:                                                   ; T=%0d", $stime);
         $display("INFO END_SIM: Simulation completed without Errors; ERRORS_COUNT=%0d", count, $stime);
        end
 endfunction
```

Figure 26 PASS criteria

The status of the test, whether it is failed or passed, is displayed in the figure that is located above. A uvm report of a test case was used to generate the figure that can be seen above.

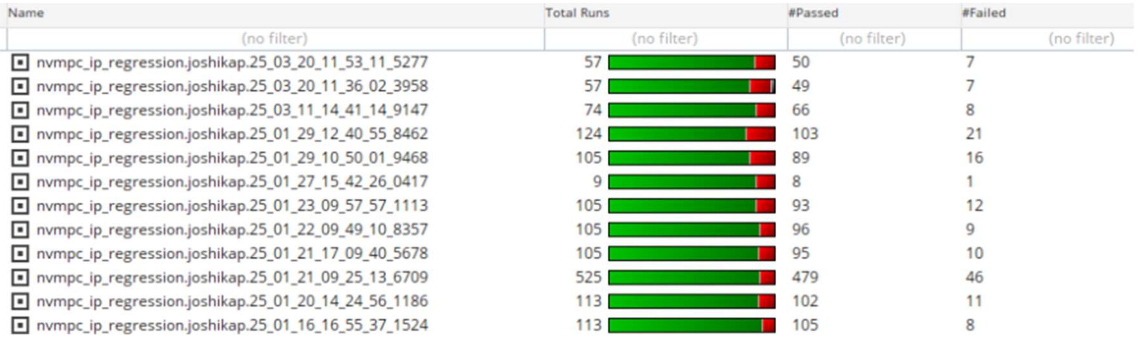
```
--- UVM Report catcher Summary ---

Number of demoted UVM_FATAL reports   :    0
Number of demoted UVM_ERROR reports   :    0
Number of demoted UVM_WARNING reports:    0
Number of caught UVM_FATAL reports    :    0
Number of caught UVM_ERROR reports    :    0
Number of caught UVM_WARNING reports :    0

--- UVM Report Summary ---

Quit count :      0 of    10
** Report counts by severity
UVM_INFO :54577
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
```

Figure 27 UVM Report summary

6.11 Regression report

All of the test cases are compiled into a single file when you perform regression analysis. The next step is to execute each of them simultaneously during the run phase of the process. This is done to check coverage and to determine whether or not the test cases were successful. Not only does it have a coverage matrix that has the svseed value, but it also contains all of the information regarding the current configuration. In the event that a testcase fails for no obvious reason but passes when it is

executed in standalone mode, we make use of the same svseed to execute the testcase once more locally and investigate the reasons under which it failed.



Name	Total Runs	#Passed	#Failed
(no filter)	(no filter)	(no filter)	(no filter)
nvmpc_ip_regression.joshikap.25_03_20_11_53_11_5277	57	50	7
nvmpc_ip_regression.joshikap.25_03_20_11_36_02_3958	57	49	7
nvmpc_ip_regression.joshikap.25_03_11_14_41_14_9147	74	66	8
nvmpc_ip_regression.joshikap.25_01_29_12_40_55_8462	124	103	21
nvmpc_ip_regression.joshikap.25_01_29_10_50_01_9468	105	89	16
nvmpc_ip_regression.joshikap.25_01_27_15_42_26_0417	9	8	1
nvmpc_ip_regression.joshikap.25_01_23_09_57_57_1113	105	93	12
nvmpc_ip_regression.joshikap.25_01_22_09_49_10_8357	105	96	9
nvmpc_ip_regression.joshikap.25_01_21_17_09_40_5678	105	95	10
nvmpc_ip_regression.joshikap.25_01_21_09_25_13_6709	525	479	46
nvmpc_ip_regression.joshikap.25_01_20_14_24_56_1186	113	102	11
nvmpc_ip_regression.joshikap.25_01_16_16_55_37_1524	113	105	8

Figure 28 Regression results

6.12 Coverage Report

In coverage report we can see the code coverage, since functional coverage is not targeted in this IP. In this we can see different type of code coverage and its percentage. Code coverage must be 100%. Achieving 100% code coverage means every line, branch, toggle, condition, state, and transition in the RTL has been exercised. This is critical for safety-critical IPs (e.g., automotive IP like NVMPC) to ensure no untested code paths remain, reducing the risk of latent bugs.

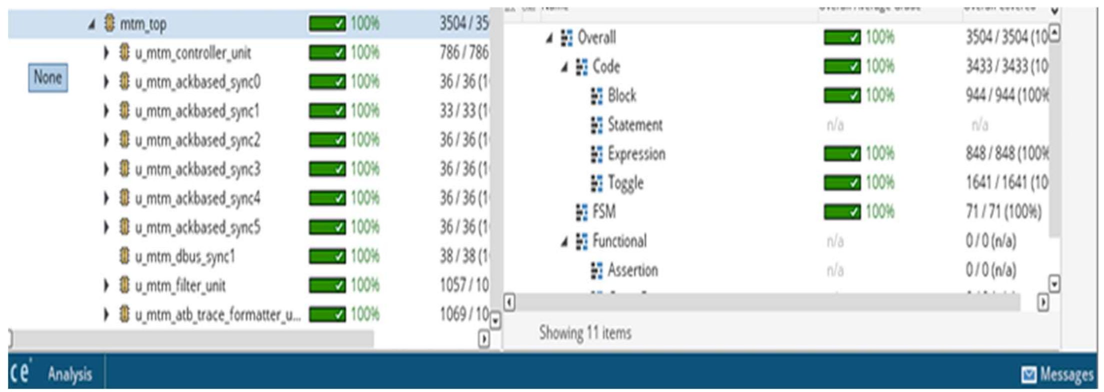


Figure 29 Coverage report 100%

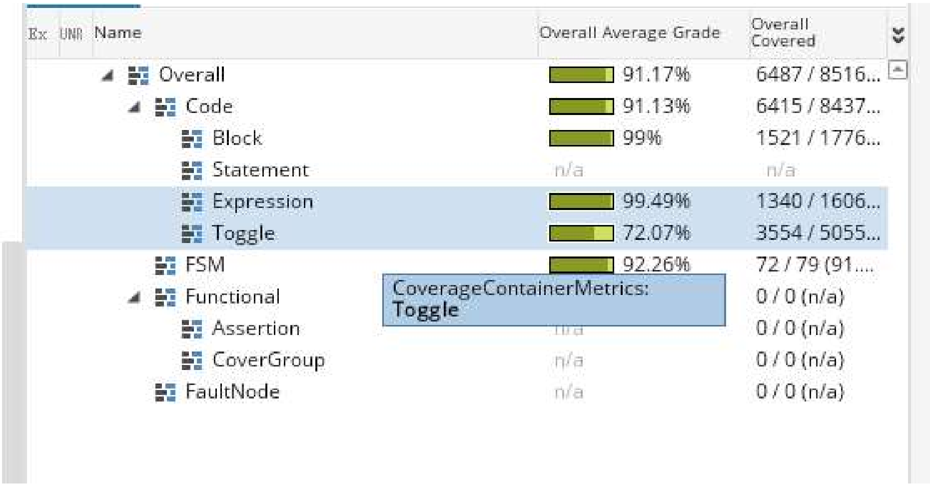


Figure 30 : Code coverage report

6.13 Summary

Following responsibilities are assigned to me during my internship period for this IP.

1. Generation of various test scenarios.
2. Generation of corner cases.
3. Vplan creation and updation
4. Regression and debugging
5. Coverage analysis
6. Documentation

Chapter 7: SoC (System on Chip) Methodology

A System on Chip (SoC) is an integrated circuit that combines key components like the CPU, memory, and peripheral interfaces onto a single silicon chip. This integration helps reduce size, power consumption, and cost while improving performance. SoCs are commonly used in devices such as smartphones, tablets, and embedded systems to provide efficient and compact computing solutions.

7.1 System on Chip (SoC) Design Flow

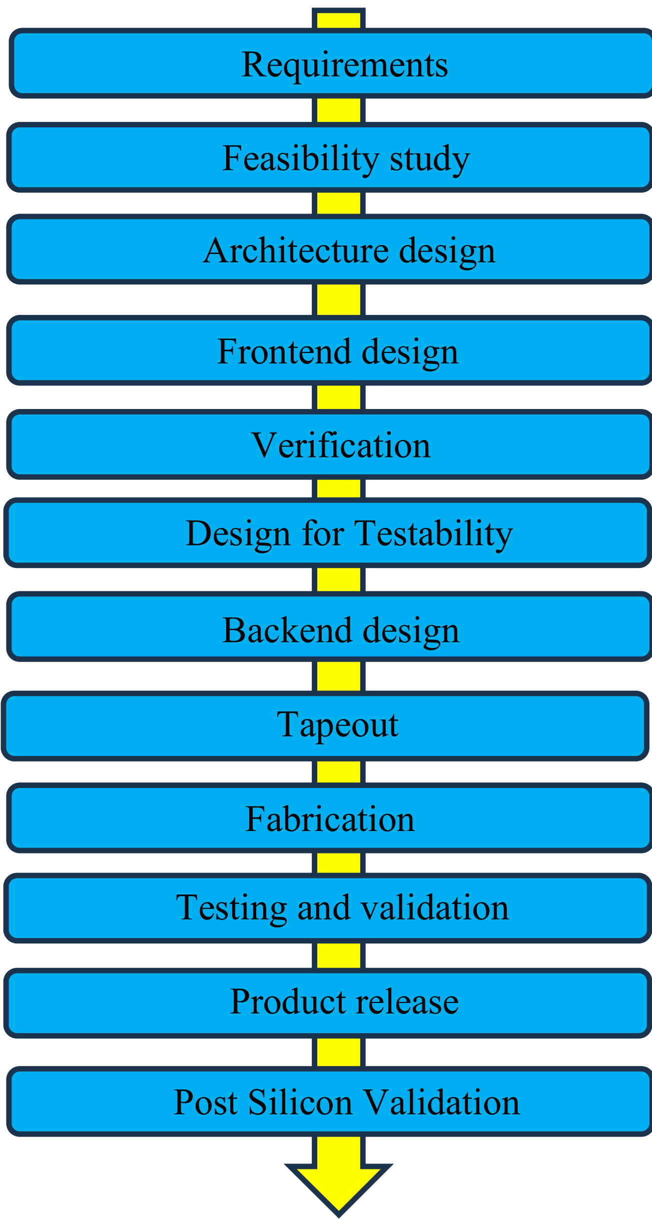


Figure 31 SoC Design Flow

7.1.1 *Requirements*

Purpose: To find out exactly what the SoC needs to do and how well it should perform.

- Study the market and competitors to see what features are needed.
- Talk to customers and users to understand their needs.
- Write down all these requirements clearly so everyone knows what the SoC should achieve.

7.1.2 *Feasibility*

Purpose: To check if the project can be done with the available technology and budget.

- Create simple designs or models to test ideas.
- Estimate how much the project will cost.
- Identify possible problems or risks and think about how to avoid them.

7.1.3 *Architecture Design*

Purpose: To plan the overall layout and structure of the SoC.

- Draw diagrams showing the main parts of the chip and how they connect.
- Decide how these parts will communicate with each other.
- Choose ready-made building blocks (IP cores) to use in the design.
- Think about power use, security, and future upgrades.

7.1.4 *Frontend Design*

Purpose: To translate the architectural plan into a detailed hardware design by writing and integrating hardware description code.

- Write detailed hardware code describing how each part functions.
- Customize and integrate the selected building blocks.
- Assemble all parts into a complete SoC design.
- Perform code checks to ensure quality and correctness.
- Set design goals like speed and power, and run early tests to verify them.

7.1.5 *Verification*

Purpose: To make sure the design works correctly and meets all the requirements.

- Run simulations to test the design's behavior under different conditions.

- Use formal verification methods to mathematically prove correctness.
- Employ emulation techniques to test the design on hardware platforms.

7.1.6 *Design for Testability*

Purpose: To add features that make it easier to test the SoC after it is built.

- Insert scan chains to help check internal circuits.
- Implement Built-In Self-Test (BIST) to enable the chip to test itself.
- Use boundary scan techniques to test connections between chips.

7.1.7 *Backend Design*

Purpose: To transform the RTL code into a physical chip layout.

- Plan the chip's floorplan by organizing major blocks and resources.
- Perform placement and routing to position components and connect them with wires.
- Conduct timing analysis to ensure signals travel within required time limits.

7.1.8 *Tapeout*

Purpose: To prepare the final design files needed for manufacturing the chip.

- Perform Design Rule Checking (DRC) to verify layout meets manufacturing rules.
- Run Layout Versus Schematic (LVS) checks to confirm the layout matches the design.
- Generate GDSII files, the standard format for chip fabrication.

7.1.9 *Fabrication*

Purpose: To manufacture the physical SoC chip.

- Produce silicon wafers where the chips are built.
- Cut the wafers into individual chips (dies).
- Package the chips to protect them and enable connection to other devices.

7.1.10 *Testing and Validation*

Purpose: To confirm that the manufactured SoC works correctly and meets performance standards.

- Perform electrical tests to check for defects.
- Conduct functional tests to verify all features work as intended.
- Run reliability tests to ensure long-term performance and durability.

7.1.11 *Product Release*

Purpose: To introduce the SoC to the market and support customers.

- Prepare and distribute detailed documentation.
- Provide customer support for integration and troubleshooting.
- Execute marketing activities to promote the product.

7.1.12 *Post–Silicon Validation*

Purpose: To verify the SoC’s performance in real-world environments and improve it as needed.

- Conduct field testing in actual use conditions.
- Debug any issues found after manufacturing.
- Release updates and patches to fix problems or enhance functionality.

Chapter 8: Conclusion

This thesis talks about the important need for strong verification methods to make sure that the Non-Volatile Memory Platform Controller (NVMPC) IP is safe, reliable, and follows the rules in automotive systems. As car electronics get more complicated and important for safety, it is important to follow ISO 26262 standards to avoid failures that could put people in danger. The study created a full verification environment that used advanced simulation methods, the Universal Verification Methodology (UVM), formal verification, and automated compliance monitoring to meet strict functional safety standards.

The work shows that the NVMPC IP can reliably handle memory operations while meeting Automotive Safety Integrity Level (ASIL) standards. This was done through careful planning of verification, development of a testbench, and thorough testing that included coverage analysis, regression, and fault injection. Adding reusable verification parts and automation tools makes verification faster and more scalable, which helps with problems caused by changing SoC architectures and the needs of the automotive industry.

The thesis also describes a systematic flow for designing and verifying SoCs, stressing how important it is for hardware and software teams to work together to ensure functional safety. The future scope shows ways to make dynamic coverage analysis better, make fault tolerance better, and keep high quality and adapt to new automotive standards through continuous integration.

Overall, this work makes a big difference in improving the verification of automotive IPs, which will help make vehicle electronics safer and more reliable in a world where cars are becoming more connected and self-driving.

Chapter 9: Future scope

The work that went into checking the Non-Volatile Memory Platform Controller (NVMPC) IP has made it safer and more reliable for cars. There are many ways to make the NVMPC better in the future so that it works better with modern car systems.

- Better Verification methods:
  In the future, the NVMPC should be checked with more advanced testing methods.
  This includes testing all parts of the design with dynamic coverage analysis and using a mix of different testing methods to find problems that aren't obvious. These methods will help you find errors that are hard to find or very rare.

- Reusable and Modular Test Components
  Test parts that can be used again and again and are modular. Making standard, reusable parts for testing will speed up and simplify the verification process. You can quickly change modular testbenches to fit new designs or changes, which saves time and effort.

- Automation and Continuous Testing:
  Automating the process of checking compliance with the safety standards and running the tests continuously will help find the issues early. This will make things better and cut down on the time it takes to release new versions of the IP.

- Stronger Error Handling:
  Making the NVMPC better at fixing mistakes and handling faults will make it more resistant to data errors and failures. This will help it meet higher safety standards and keep car systems safe.

- Clear Documentation and Keeping Up with Standards
  Keeping detailed documentation and sharing best practices will help teams learn and improve over time. Also, staying updated with new safety standards and considering power and performance needs will ensure the NVMPC stays reliable and efficient

References

[1] M. J. Prajwala, K. Desai, and L. He, "Verification of NAND Flash Controller," Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering 2022, 6-8 July, 2022, London, U.K., 2022, pp. 70-76.

[2] Y. Yun, J. Kim, N. Kim and B. Min, "Beyond UVM for practical SoC verification," 2011 International SoC Design Conference, Jeju, 2011.

[3] A. Ismail, Q. Liu and W. Jung, "ISO 26262 automotive functional safety: issues and Challenges," International Journal of Reliability and Applications, 2014.

[4] K.S.Pooja, S.Krishnakumar and H.V.R.Aradhva "Verification of interconnection IP for automobile applications using system Verilog and uvm," 2018 3rd IEEE International Conference on recent trends in electronics, information & communication technology, Banglore,INDIA,2018

[5] R. Madan, N. Kumar and S. Deb, "Pragmatic approaches to implement self-checking mechanism in UVM based TestBench," 2015 International Conference on Advances in Computer Engineering and Applications, Ghaziabad, 2015.

[6] Zhang, L., & Wang, H., "Optimizing Flash Controller Verification with UVM and Coverage-Driven Techniques," 2024 IEEE International Conference on Integrated Circuits and Systems (ICICS), Beijing, China, 2024.

[7] H. Zhaohui, A. Pierres, H. Shiqing, C. Fang, P. Royannez, E. P. See and Y. L. Hoon, "Practical and Efficient SOC Verification Flow by Reusing IP Testcase and Testbench," 2012.

[8] S. A. Saji and K. Sivasankaran, "Test suite for SoC interconnect verification," 2017 International Conference on Microelectronic Devices, Circuits and Systems (ICMDCS), Vellore, 2011.

[9] A. Manzone, A. Pincetti and D. De Costantini, "Fault tolerant automotive systems: an overview," Proceedings Seventh International On-Line Testing Workshop, Taormina, Italy, 2001.

[10] International Standards, "ISO 26262 Functional safety for road vehicles," Geneva, Switzerland, 2011.

[11] Choi, H., & Park, J., "UVM-Based Verification of Flash Memory with Error Correction Codes," 2024 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, 2024.

[12] M. K. Wooseung Yang, "Current Status and Challenges of SoC Verification for Embedded Systems Market," IEEE, pp. 213-216, 2003.

[13] M. Tiikkainen, "Automated Functional Coverage Driven Verification With Universal Verification Methodology," University of Oulu, Oulu, Finland, March 2017.

[14] V. S. Rashmi, G. Somayaji and S. Bhamidipathi, "A methodology to reuse random IP stimuli in SoC functional verification environment," 2015 19th International Symposium on VLSI Design and Test, Ahmedabad, 2015.

[15] R. Wang, W. Zhan, G. Jiang, M. Gao and S. Zhang, "Reuse issues in SoC verification platform," 8th International Conference on Computer Supported Cooperative Work in Design, Xiamen, China, 2004.

[16] F. A. da Silva et al., "Determined-Safe Faults Identification: A step towards ISO26262 hardware compliant designs," 2020 IEEE European Test Symposium (ETS), Tallinn, Estonia, 2020.

[17] A. B. Mehta, "ASIC/SoC Functional Design Verification: A Comprehensive Guide to Technologies and Methodologies," Springer International Publishing AG, 2018.

[18] Jain and R. Gupta, "Scaling the UVM_REG Model towards Automation and Simplicity of Use," 2015 28th International Conference on VLSI Design, Bangalore, 2015.

[19] IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012), "IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language," IEEE, 2018.

[20] F. A. da Silva, A. C. Bagbaba, S. Hamdioui and C. Sauer, "Efficient Methodology for ISO26262 Functional Safety Verification," 2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS), Rhodes, Greece, 2019.

[21] J. Doe, J. Smith, and M. Johnson, "Fault Injection Techniques for Robust System Verification: A Comprehensive Review," Journal of System Verification, vol. 12, no. 3, pp. 45-67, 2023.

[22] Salah, K., "A Unified UVM Architecture for Flash-Based Memory Verification," IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 70, no. 4, pp. 1234-1240, 2023.

[23] Nguyen, T. H., & Tran, V. D., "Verification of Flash Memory Controller Using UVM with SystemVerilog Assertions," 2023 IEEE International Conference on Electronics, Information, and Communication (ICEIC), Singapore, 2023.

[24] Patel, R., & Sharma, A., "Design and Verification of a High-Performance NAND Flash Controller Using SystemVerilog and UVM," 2023 IEEE International Symposium on Circuits and Systems (ISCAS), Monterey, CA, 2023.

[25] Kim, J., & Lee, S., "SystemVerilog-Based UVM Testbench for Flash Memory Interfaces," 2023 IEEE Asian Solid-State Circuits Conference (A-SSCC), Seoul, South Korea, 2023.

[26] Zhang, L., & Wang, H., "Optimizing Flash Controller Verification with UVM and Coverage-Driven Techniques," 2024 IEEE International Conference on Integrated Circuits and Systems (ICICS), Beijing, China, 2024.

[27] Li, X., & Chen, Y., "SystemVerilog UVM Framework for Flash Memory BIST Verification," 2024 IEEE European Test Symposium (ETS), Tallinn, Estonia, 2024.

[28] Ahmed, M., & Khan, F., "High-Speed Flash Controller Design and UVM Verification," 2023 IEEE International Conference on Computer-Aided Design (ICCAD), San Francisco, CA, 2023.

[29] Srivastava, P., & Jain, V., "Verification of Flash Memory Controllers Using UVM and Formal Methods," 2024 IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS), Rhodes, Greece, 2024.

[30] Wu, Q., & Liu, Z., "SystemVerilog UVM Testbench for Low-Power Flash Memory Controllers," 2023 IEEE International Conference on Microelectronics (ICM), Abu Dhabi, UAE, 2023.

Dr. Geetika DUA

Kapil Joshi_plag_check_22

me thesis

Document Details

Submission ID

trn:oid:::3618:104784746

Submission Date

Jul 17, 2025, 12:11 PM GMT+5:30

Download Date

Jul 17, 2025, 12:13 PM GMT+5:30

File Name

Kapil Joshi_plag_check_22.pdf

File Size

3.4 MB

41 Pages

8,757 Words

54,639 Characters

5% Overall Similarity

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

Match Groups

- **40** Not Cited or Quoted 5%
  Matches with neither in-text citation nor quotation marks
- **0** Missing Quotations 0%
  Matches that are still very similar to source material
- **1** 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

1% Internet sources
0% Publications
4% Submitted works (Student Papers)

Integrity Flags

0 Integrity Flags for Review

Our system's algorithms look deeply at a document for any inconsistencies that would set it apart from a normal submission. If we notice something strange, we flag it for you to review.

A Flag is not necessarily an indicator of a problem. However, we'd recommend you focus your attention there for further review.