Design & Verification of Memory Protection Unit in Automotive Microcontroller

M.Tech. Thesis

By

AASTHA JHA

DISCIPLINE OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY INDORE

June 2023

Design & Verification of Memory Protection Unit in Automotive Microcontroller

M.Tech. Thesis

A THESIS

Submitted in partial fulfillment of the
requirements for the award of the degree
of
Master of Technology

by

AASTHA JHA

DISCIPLINE OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY INDORE

June 2023

INDIAN INSTITUTE OF TECHNOLOGY INDORE

CANDIDATE'S DECLARATION

I hereby certify that the work which is being presented in the thesis entitled **Design & Verification of Memory Protection Unit in Automotive Microcontroller** in the partial fulfillment of the requirements for the award of the degree of **MASTER OF TECHNOLOGY** and submitted in the **DISCIPLINE OF ELECTRICAL ENGINEERING, Indian Institute of Technology Indore**, is an authentic record of my own work carried out during the time period from June 2022 to June 2023 under the supervision of Prof. Santosh Kumar Vishvakarma, Professor, Indian Institute of Technology Indore.

The matter presented in this thesis has not been submitted by me for the award of any other degree of this or any other institute.

Aastha Jha
26/05/2023
(Aastha Jha)

This is to certify that the above statement made by the candidate is correct to the best of my/our knowledge.

26/05/2023
(Prof. Santosh Kumar Vishvakarma)

Aastha Jha has successfully given his/her M.Tech. Oral Examination held on **12/05/2023**.

Signature(s) of Supervisor(s) of M.Tech. thesis
Date: 26/05/2023

Convener, DPGC
Date:

Signature of PSPC Member #1
Date:

Signature of PSPC Member #2
Date:

ACKNOWLEDGEMENTS

I would like to take this moment to convey my appreciation to everyone who helped make this period learnable, enjoyable, and pleasant. First and foremost, I'd want to express my gratitude to my supervisor, **Prof. Santosh Kumar Vishvakarma**, who was a continual source of inspiration during my research. This research activity was conducted with his ongoing mentoring and research suggestions. His unwavering support and encouragement have inspired me to stay focused on my studies.

I am Very Grateful for my Team and my Mentor **Mr. Saransh Mehrotra** for the support and Guidance and thanks for allowing me to present the Real work.

I would also like to thank **Prof. Mukesh Kumar** and **Prof. Hem Chandra Jha**, members of my research progress committee, for taking the time to assess my progress throughout the semesters. Their insightful feedback and ideas assisted me in improving my work at various levels.

I would want to express my gratitude to Indian Institute of Technology Indore for allowing me to test my research talents. My sincere acknowledgement to all members of the Nanoscale Devices, VLSI Circuit System Design Lab (NSDCS) research group, particularly **Dr. Gopal Raut** and **Mr. Narendra Singh Dhakad** for their discussions and assistance throughout my thesis study.

Special Thanks to STMicroelectronics Pvt. Ltd. for providing opportunity to build my career as a Design Verification Engineer.

I would like to express my heartfelt respect to my family for their love, care and support they have provided to me throughout my life.

Aastha Jha

This Thesis is Dedicated

to

The Almighty GOD,
My Family, and Friends

ABSTRACT

The verification of the AXI (Advanced eXtensible Interface) protocol and MPU (Memory Protection Unit) in automotive microcontrollers is of paramount importance to ensure the reliability, safety, and security of these systems in the context of Advanced Driver Assistance Systems (ADAS).

The AXI protocol, a popular interface in automotive microcontrollers, makes it easier for different system peripherals and components to communicate with one another. The AXI protocol's compliance with specifications and industry standards is verified using verification techniques such simulation, formal verification, and protocol compliance testing.

On the other hand, the MPU is in charge of implementing memory access regulations and safeguarding private information and resources in the microcontroller. To make sure that the MPU checks are appropriately implemented and successfully enforce memory protection, it is verified through extensive testing, assertion-based verification, and coverage analysis. The verification procedure aids in locating and resolving problems including data corruption, unauthorised access, and boundary violations. The AXI protocol and MPU verification have a big impact on ADAS systems. Microcontrollers play a significant role in the management of crucial ADAS processes, including sensor data processing, decision-making, and actuator control.

By successfully verifying the AXI protocol and MPU in automotive microcontrollers, potential issues and vulnerabilities can be identified and addressed early in the development cycle. This proactive approach improves the overall reliability, performance, and safety of ADAS systems, reducing the risk of failures, malfunctions, and security breaches in real-world scenarios.

Contents

Abstract **i**

List of Figures **v**

List of Abbreviations **vii**

1 Introduction **1**

1.1 Overview 1

1.2 ADAS: Advanced Driver Assistance System 2

1.2.1 ADAS Applications 3

1.3 Automotive Microcontroller 7

1.3.1 Basic Components of Automotive Micro-controller 8

1.4 AMBA Protocols 10

1.5 Literature Survey 12

1.6 Objectives 15

1.7 Organization of Thesis 15

2 Design Verification of AXI Protocol **17**

2.1 Overview of the AXI Protocol 17

2.2 Design of AXI protocol 18

2.2.1 Master-slave communication: 19

2.2.2 AXI transaction types: 20

2.3 Verification of AXI Protocol 21

2.4 Summary 23

3 Design of Memory Protection Unit **25**

3.1 Introduction to Memory Protection Unit 25

3.2 Block Design of MPU 26

3.2.1 Snippet of Verilog Code for MPU Design 28

3.2.2 IP Wrapper of MPU 29

3.2.3 Waveform results of MPU design 30

3.3 Summary 31

4 Verification of Memory Protection Unit **33**

4.1 Introduction to Verification of MPU 34

4.2 Functional Verification of MPU Checks 36

4.2.1 Virtual Machine matched or mismatched check 36

4.2.2 Memory-type matched Or mismatched check 41

4.2.3 Access type (Read/Write/Execute) check 43

4.2.4 Multiple regions access check 46

4.2.5 Boundary cross check 49

4.3 Formal Verification of MPU 52

4.3.1 Assertions for MPU Checks 52

4.3.2 Assertions Results 53

4.3.3 Coverage Results 54

4.4 Summary 56

5 Conclusion **57**

5.1 Work Conclusion 57

5.2 Future Scope of Work 58

References **61**

List of Figures

1.1 ADAS Applications 4
1.2 ADAS 7
1.3 Basic Structure of Automotive Microcontroller 8
1.4 AMBA Architecture 11

2.1 AXI Protocol Channels 19
2.2 AXI Functional Verification Waveform 22

3.1 Memory Protection Unit 26
3.2 MPU Block Design 27
3.3 IP wrapper of MPU 30
3.4 Waveform results of MPU Design 31

4.1 Verification Phases 34
4.2 Functional Verification 36
4.3 SoC interaction with Operating Systems 37
4.4 Virtual Machine Matched and Mismatched 38
4.5 Virtual Machine Matched Waveform 39
4.6 Virtual Machine Mismatched Waveform 40
4.7 Memory-type Matched Or Mismatched 41
4.8 Memory-type Matched Or Mismatched 43
4.9 Access Type (Read/Write/Execute) 44
4.10 Access Type (Read/Write/Execute) 46
4.11 Multiple Regions Access 47

4.12 Multiple Regions Access 48
4.13 Boundary Cross check 49
4.14 Boundary Cross check 51
4.15 Assertions Results 54
4.16 Coverage Results 54
4.17 Modified Coverage Results 55

List of Abbrevations

ADAS	:	Advanced Driver Assistance System
AMBA	:	Advanced Micro controller Bus Architecture
AHB	:	Advanced High-performance Bus
APB	:	Advanced Peripheral Bus
ASB	:	Advanced System Bus
AXI	:	Advanced eXtensible Interface
DMA	:	Direct Memory Access
IP	:	Intellectual Property
MPU	:	Memory Protection Unit
MCU	:	Micro Controller Units
NVM	:	Non-Volatile Memory
NoC	:	Network-On-Chip
OS	:	Operating System
ROM	:	Read-Only Memory
SoC	:	System on Chip
SRAM	:	Static Random Access Memory
VM	:	Virtual Machines
UVM	:	Universal Verification Methodology

Chapter 1

Introduction

1.1 Overview

This report provides a brief overview of the verification process for the AXI protocol and MPU in automotive microcontrollers, with a specific focus on their relevance in the context of Advanced Driver Assistance Systems (ADAS). The report highlights the significance of verification techniques such as functional verification, formal verification, and compliance testing in ensuring the reliability, safety, and security of ADAS systems. By verifying the AXI protocol, potential issues such as protocol violations, timing problems, and interoperability challenges can be identified and addressed, thereby improving the overall performance and compliance of the system. Similarly, the verification of the MPU checks ensures the enforcement of memory access rules and protection of sensitive data, mitigating risks related to unauthorized access, boundary crossings, and data corruption. The impact of these verification processes on ADAS is substantial, as they contribute to the trustworthiness, functionality, and compliance of the microcontrollers used in ADAS applications. Through thorough verification, potential issues are identified early, leading to enhanced reliability and safety in real-world ADAS scenarios.

1.2 ADAS: Advanced Driver Assistance System

Human error is to blame for the majority of automobile collisions, which can be prevented by using an ADAS system. Through a decrease in traffic accidents and a lessening of their negative effects, ADAS seeks to avoid fatalities and injuries. A variety of electronic technologies are referred to as advanced driver-assistance systems (ADAS) and are used to help drivers with driving and parking tasks. By establishing a safe interaction between people and machines, it aims to improve auto and road safety. To detect barriers in the immediate area and pinpoint driver faults, ADAS makes use of automated technologies such as vehicle sensors and cameras. It then takes the proper action to ensure safety. Depending on the components built into the car, the degree of autonomous driving varies. Since human error causes the bulk of traffic accidents, ADAS is intended to automate, adapt, and enhance vehicle technology to improve safety and driving. ADAS has been shown to reduce traffic fatalities by minimising human error. By warning drivers of threats, putting protective measures in place, and assuming control when necessary, ADAS safety features assist in preventing crashes and accidents. These options may consist of adaptive lighting, adaptive cruise control, collision avoidance systems, satellite navigation, traffic warnings, obstacle alerts, lane departure and centering aids, as well as traffic warnings and traffic warning systems.In 2021, a report by Canalys found that ADAS features were present in about 33 percent of new cars sold in the US, Europe, Japan, and China. The research also predicted that by 2030, 50 percent of all on-road vehicles would be equipped with ADAS. Automobiles will become increasingly important in the next wave of mobile-connected devices as autonomous vehicles quickly emerge. System-on-chips (SoCs) are solutions for autonomous applications that connect sensors and actuators through interfaces and high-performance electronic controller units (ECUs). To achieve a 360-degree field of view, self-driving cars use a variety of near and far-field technologies and applications. In order to match rising performance and physical footprint standards while consuming less power, hardware designs are adopting more cutting-edge manufacturing nodes.

1.2.1 ADAS Applications

Prior to airbags and three-point seat belts, passive safety measures such as shatter-resistant glass and airbags were the main focus of developments in automobile safety. But with the addition of embedded vision, ADAS systems now actively improve safety by lowering the frequency of collisions and occupant injuries. Vehicle cameras have made it necessary to create a new AI function that uses sensor fusion to recognise and analyse objects. Similar to how the human brain analyses information, sensor fusion combines large volumes of data using image recognition software, ultrasonic sensors, lidar, and radar. This technology enables real-time review of streaming video, object detection, and decision-making because it can physically react more quickly than a human driver. Several of the most widespread ADAS applications include:

1. Adaptive Cruise Control: On highways, adaptive cruise control (ACC) has a lot of advantages since it makes it easier for drivers to maintain speed and keep an eye on other vehicles for extended periods of time. ACC can automatically change the vehicle's speed, decelerate, and even come to a complete stop when necessary by analysing the movements of surrounding objects.

2. Glare-Free High Beam and Pixel Light: Modern lighting innovations like Pixel Light and Glare-Free High Beam greatly increase nighttime visibility and put driving safety first. The problem of standard high beams blinding approaching vehicles is addressed with glare-free high beams, also known as adaptive high beams or matrix high beams. With this clever adjustment, drivers can take advantage of high beams' improved vision without jeopardising other drivers' safety. On the other hand, Pixel Light is a cutting-edge lighting system that uses a variety of unique LED components to provide accurate and adaptive illumination. Pixel Light improves visibility, lowers glare, and ultimately increases overall driving safety by offering focused and effective lighting.

3. Adaptive Light Control: A technology known as adaptive light control allows a car's headlights to automatically adjust to the ambient lighting. It dy-

namically modifies the headlights' intensity, direction, and alignment based on the environment's conditions and degree of darkness.

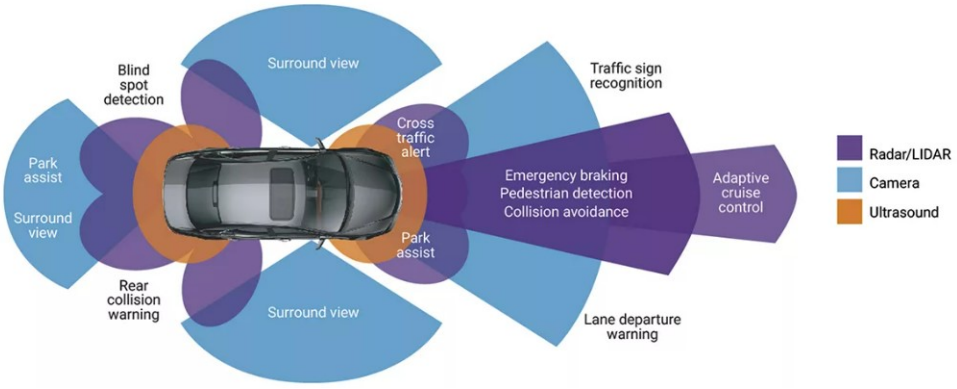

Figure 1.1: ADAS Applications

4. Automatic Parking: Automatic parking systems alert drivers to any blind areas, which they can use to decide whether to move the steering wheel or not and when to stop. Compared to vehicles with traditional side mirrors, those with rearview cameras give drivers a greater sense of their surroundings. By merging the data from several sensors, advanced systems can even complete parking moves without the driver's direct involvement.

5. Autonomous Valet Parking: Blind spots are alerted to drivers by automatic parking systems, which helps them determine whether to move the steering wheel and when to stop. The vision from a vehicle's rearview camera is superior to that of a vehicle's standard side mirrors. Using information from several sensors, sophisticated systems can even complete parking moves without the driver's direct involvement.

6. Navigation System: To help drivers navigate a route while keeping their attention on the road, car navigation systems provide voice prompts and on-screen cues. Additionally, some navigation systems have the capacity to show current traffic data and, if necessary, recommend other routes to avoid gridlock.

Furthermore, in order to help drivers stay focused on the road, modern navigation systems may include Heads-up Displays (HUDs).

7. Night Vision: Drivers may now see objects that would normally be difficult or impossible to notice at night thanks to night vision technology. There are two types of night vision technology: passive night vision systems and active night vision systems. The thermal energy that is emitted by numerous things, including vehicles, animals, and other environmental factors, is what passive night vision devices use to operate. These technologies produce a visual depiction by using the variations in heat signatures.

8. Blind Spot Monitoring: Drivers can get important information from blind spot detection systems that would be difficult or impossible to get any other way. When a vehicle is trying to change lanes into an inhabited area, for example, these sensors are vital in spotting objects that are in their blind spot. Certain systems are made to go into alarm mode in such circumstances, warning the driver of the potential risk.

9. Automatic Emergency Braking: In order to avoid collisions with other vehicles or obstructions on the road, automatic emergency braking systems use sensors. When a risk is discovered, these technologies are able to analyse the immediate environment and warn the driver in a timely manner. Certain emergency braking systems can implement proactive safety measures such as automatically tightening seat belts, lowering vehicle speed, and applying adaptive steering strategies in order to avoid an accident. A potential collision is avoided or its effects are reduced by using these preventive measures.

10. Crosswind Stabilization: Vehicles are supported by the most recent ADAS function when they come into heavy crosswinds. This technology can detect high pressure being applied to the car while it is moving and react by applying the brakes to the wheels being affected by the crosswinds. This action assists in stabilising the car and reducing the force of the crosswind.

11. Driver Drowsiness Detection: Systems for detecting driver drowsiness are created to spot indicators of sleepiness or other driving issues and send timely alerts to the driver. The driver's degree of concentration is measured using a variety of techniques. As an illustration, sensors can examine the driver's head motions

and heart rate to detect indicators of sleepiness. In order to keep the driver alert and focused on the road, some systems also send driving alerts that resemble lane departure warning signs. By using these methods, driver drowsiness monitoring systems help to improve driver safety and reduce accidents that result from distracted driving.

12. Driver Monitoring System: Another reliable way to assess a motorist's attentiveness is to use a driver monitoring system. This technology uses video sensors to analyse whether the driver's eyes are fixed on the road or if they are prone to drifting. Drivers can be alerted to potential distractions using a variety of alert techniques, including aural messages, steering wheel feedback, or visual indicators like flashing lights. In extreme circumstances, the system might even stop the car entirely in order to keep everyone safe. The installation of a driver monitoring system offers an additional level of watchfulness to encourage focused and responsible driving.

13. 5G and V2X: With greater reliability and reduced latency, this cutting-edge 5G ADAS technology delivers enhanced V2X (vehicle-to-everything) communication capabilities, enabling seamless interaction between vehicles, pedestrians, and infrastructure. As cellular networks are now used by millions of automobiles, this feature revolutionises real-time navigation by delivering greater situational awareness. It makes it easier to keep an eye on traffic conditions and adjust speed accordingly, and it provides real-time GPS map updates via established channels and cellular networks.

A growing number of software-driven components in automobiles require over-the-air (OTA) software updates, which can only be made possible by a V2X connection. This contains essential updates, including map improvements, bug repairs, security improvements, and more. Significant modifications in the automobile design process are necessary as automotive electronic hardware and software develop in order to solve the convergence of competing goals:

1. Improving dependability 2. Cost-cutting 3. Cutting down on development cycles

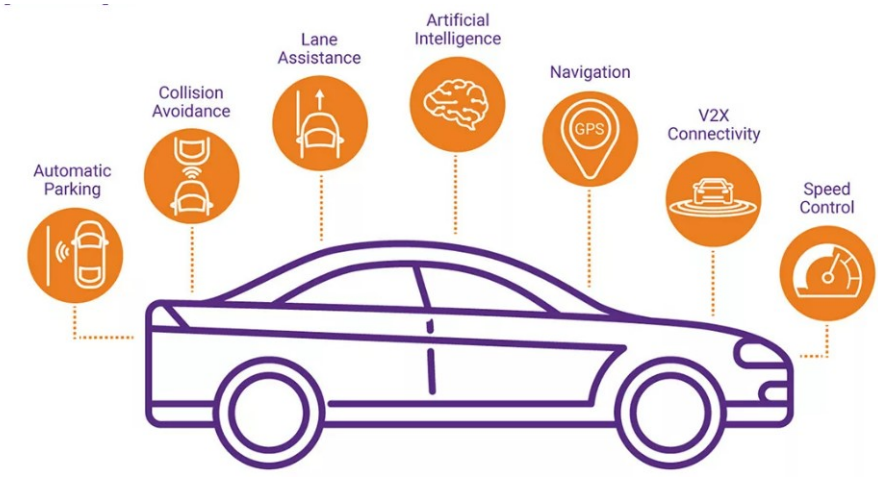

Figure 1.2: ADAS

A move away from decentralised ADAS domain controllers and towards centralised ADAS electronic controller units (ECUs) is now taking place in the automotive industry. The need for fully autonomous vehicles that can independently sense their environment and function without human interference is what is causing this transformation. As a result, to accommodate this developing technology, automobiles' electrical designs must be upgraded. The amount of data processed grows in direct proportion to advancements in electronic design. The newly integrated domain controllers need better processing performance, less power, and smaller packaging to handle this data. The use of 64-bit CPUs, neural networks, and AI accelerators requires the use of cutting-edge semiconductor features, semiconductor process technologies, and connectivity technologies in order to enable enhanced ADAS capabilities. Centralised computing systems become increasingly important as the number of electronic components decreases, enabling the efficient and effective integration of ADAS features.

1.3 Automotive Microcontroller

Due to the rising complexity of contemporary automobiles, the introduction of ADAS has resulted in an increased reliance on automotive microcontrollers. The

average number of microcontroller units (MCUs) found in modern medium-class cars is between 50 and 60. These MCUs are used for a variety of functions, including autonomous driving capabilities, powertrain control, brakes, radar systems, and engine management.

A quiet revolution in automotive technology, notably in the area of in-vehicle networking, has been sparked by the development of MCUs. This development has made it possible to do away with the bulky wiring harnesses that were previously required for control circuits. The entire system architecture has been simplified by the incorporation of MCUs, enabling more effective and complex control and communication inside the vehicle.

1.3.1 Basic Components of Automotive Micro-controller

Several masters, slaves, different AMBA protocol buses, a network on chip (NOC), and a memory protection unit (MPU) are among the basic components of an automotive microcontroller (shown in Figure 1.3). Together, these parts provide smooth operation and improved functionality inside the microcontroller architecture.

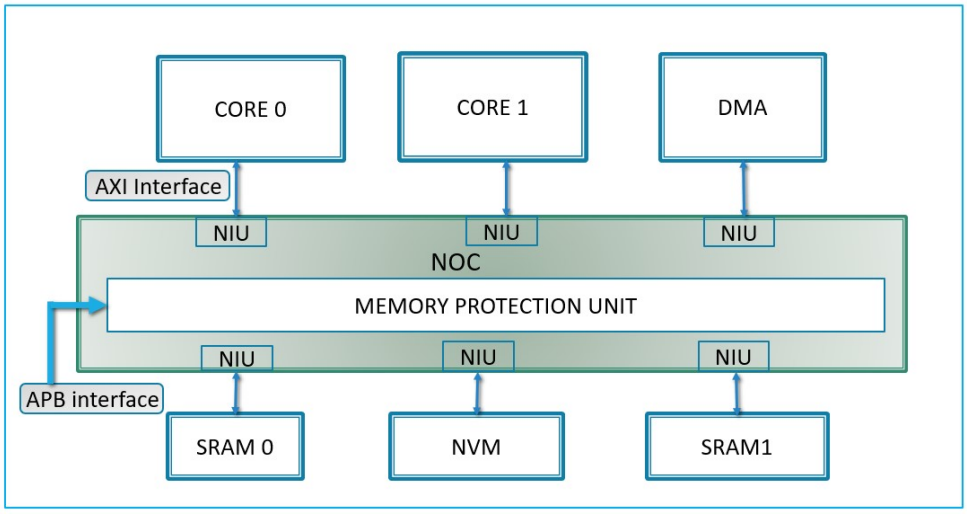

Figure 1.3: Basic Structure of Automotive Microcontroller

Let's dive into each of them in detail:

1. Masters: In the domain of electronic hardware, masters exert direct control over one or more devices, acting as the controlling bodies and the targeted devices as the controlled parties. Depending on the exact circumstance or need, the master gadgets might also serve as slaves. Devices that frequently act as masters within a system include several cores (processors) and DMA (Direct Memory Access).

2. Slaves: In the context of hardware components, slaves are things that have a master over them. Multiple slaves can be commanded at once by a single master. The terms RAM (Random Access Memory), ROM (Read-Only Memory), NVM (Non-volatile memory), and others are frequently used to describe slaves. Under the direction and control of the controlling master device, these physical components function.

3. Bus Architecture: A key component of efficient on-chip communication and controlling functional blocks in system-on-a-chip (SoC) designs is the open standard known as the ARM Advanced Microcontroller Bus Architecture (AMBA). It offers a uniform foundation for smooth connectivity and coordination across different components.

To meet various purposes, various AMBA protocols have been created, including:

APB (Advanced Peripheral Bus): This protocol is especially made to link low-bandwidth peripherals to the system, ensuring effective and dependable communication.

AHB (Advanced High-performance Bus): AHB provides a high-performance bus architecture that enables effective data transfer and enhances overall system performance [1].

ASB (Advanced System Bus): ASB offers a dependable and expandable system bus design that permits efficient communication between various system components.

AXI (Advanced eXtensible Interface): AXI is a flexible and expandable interface that satisfies the requirements of high-performance and complex systems, enabling effective data transfer and boosting system-level performance.

These AMBA protocols play a crucial role in creating effective communication

and coordination within SoC architectures, ensuring seamless interchange between various functional blocks while fulfilling the unique needs of various systems.

4. Network-on-chip (NOC): System-on-Chip (SoC) components are linked together by a Network-on-Chip (NoC), a communication subsystem based on a network architecture. Determining the best communication routes for the system's masters and slaves is a critical task it completes, ensuring effective and optimised data transport. In the SoC, the NoC serves as a central controller, intelligently controlling the communication flow between various modules to provide smooth connectivity and efficient data transmission.

5. Memory Protection Unit (MPU): Within the Network-on-Chip (NoC) architecture, the Memory Protection Unit (MPU) is a crucial hardware element that provides effective memory protection capabilities. The MPU is essential in controlling how accessible memory areas are for various masters in the system. It efficiently handles the read and write permissions given to each master and regulates which masters have access to particular memory regions. The MPU ensures secure and regulated memory access by imposing access limitations, boosting system integrity, and guarding against unauthorised access or data corruption.

1.4 AMBA Protocols

Intellectual Property (IP) is carefully developed to communicate with one another through standardised interfaces in order to guarantee seamless integration into the target system. The AMBA bus, offered by ARM, is one such commonly used interface. Traditional system-on-chip (SoC) architectures based on AMBA use the Advanced Peripheral Bus (APB) protocol for low-bandwidth peripheral interconnections and the Advanced High-Performance Bus (AHB) or Advanced System Bus (ASB) protocols for high-bandwidth interconnectivity [2]. Point-to-point connectivity is made possible by the Advanced Extensible Interface (AXI) protocol, which was added in the later AMBA 3 iteration of the specification. The features and performance of the interface were significantly improved in 2010 with the release of

an improved version dubbed AXI 4.

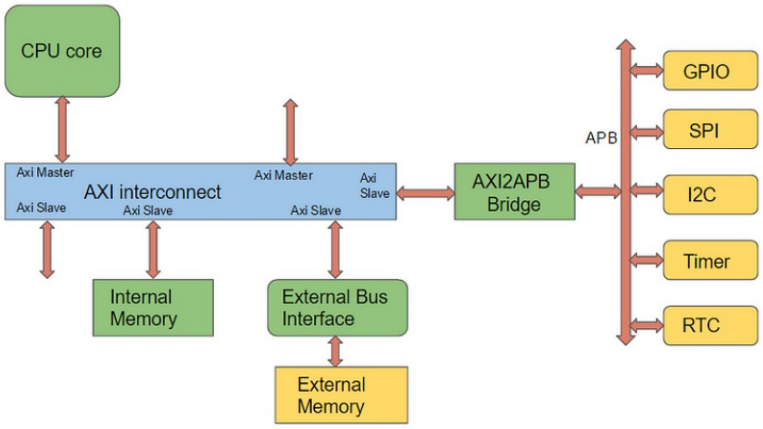

Figure 1.4: AMBA Architecture

We will focus mostly on APB and AXI protocols:

1. APB Protocol: A clear, non-pipelined interface that allows many transfer types is the APB (Advanced Peripheral Bus) protocol. Write operations with or without wait states and read operations with or without wait states are both included in these transfers. Basic Read and Write signals were a part of the APB protocol in the older AMBA 2 version. Wait transfers (PREADY) and error response (PSLVERR) were two new features added with the release of AMBA 3, more precisely APB3 (APB v1.0). Further improvements to the protocol were implemented in the most recent version, AMBA 4 or APB4 (APB v2.0), including the inclusion of protection signals (PPROT) and strobe signals (PSTRB). In order to facilitate effective peripheral communication within the system, the APB protocol has seen improvements across many AMBA versions, expanding its functionality.

2. AXI Protocol: Full AXI4, AXI-Lite, and AXI4-Stream are the three types of AXI4 interfaces that AMBA4 defines.

In order to convey data effectively, the AXI4 interface has five different channels:

Write Address channel (AW) handles the transmission of address data, when writing operations are carried out. It makes sure the target address is transmitted correctly.

Write Data Channel (W) is used to send the real data to the designated tar-

get address. By doing this, it makes sure that the right data is transmitted and synchronised with the address channel.

The Write Response channel (B): responds with information about the write operation's progress towards completion. It provides the initiator with information regarding the write transaction's success or failure.

The Read Address channel (AR): is in charge of conveying the address from which data needs to be fetched during read operations. It guarantees that the right part of memory is accessible for reading.

Via the Read Data channel (R), the requested data is transferred from the given address. In the event that the read operation is successful, it returns the data to the initiator.

These five AXI4 channels ensure efficient and reliable communication between various components in the system, enabling seamless data transfer and synchronization.

1.5 Literature Survey

Verification of MPU and AXI units have been explored in literature thoroughly. Like in [3], authors focuses on the verification of interconnection intellectual property (IP) designed for automobile applications. The authors employ System Verilog and Universal Verification Methodology (UVM) to validate the functionality and reliability of the interconnection IP. They discuss the challenges specific to automobile applications and highlight the importance of thorough verification to ensure the IP's performance in real-world automotive environments. The paper presents the design and implementation of the verification environment using System Verilog and UVM, and it discusses the test scenarios and methodologies employed. The authors evaluate the effectiveness of their approach through simulations and present the results of the verification process. The research provides insights into the verification techniques for interconnection IP in the context of automobile applications, contributing to the development of reliable and safe automotive systems. [4] pro-

vides a comprehensive guide to the various technologies and methodologies used in functional design verification of Application-Specific Integrated Circuits (ASICs) and System-on-Chip (SoC) designs. The author covers a wide range of topics, including verification planning, testbench development, simulation, assertion-based verification, hardware acceleration, and formal verification techniques. The book aims to equip readers with a deep understanding of the challenges and best practices involved in functional design verification. It serves as a valuable resource for engineers, researchers, and students in the field of ASIC and SoC design verification, offering practical insights and guidance for ensuring the correctness and reliability of complex integrated circuits. In [5], authors delve into the complexities of verifying SoC designs, highlighting the growing gap between design complexity and verification capabilities. They explore challenges such as scalability, functional correctness, power and performance verification, and the impact of emerging technologies. The paper provides valuable insights into the current state of SoC design verification and offers a glimpse into the future trends in this field. [6] describes the challenges related to efficiently supporting AXI4 transaction ordering requirements in many-core architectures. The authors propose a novel mechanism called "Group-Based Dependency Check" that enhances the AXI4 protocol to ensure correct transaction ordering in a many-core system. They introduce a lightweight hardware component called the Order Dependency Tracker (ODT) that efficiently tracks and resolves transaction dependencies. The proposed solution improves system performance by reducing the overhead of maintaining transaction order, and it achieves better scalability in large-scale many-core systems. The paper provides valuable insights and practical techniques for optimizing the AXI4 protocol in the context of many-core architectures. In [7] explores the topic of predictable virtualization on microcontrollers with a memory protection unit (MPU). The authors propose a novel approach that enables predictable and efficient virtualization on MPUs. They introduce a framework called "PROVM" that leverages the MPU to enforce memory isolation between virtual machines (VMs) while minimizing performance overhead. The paper discusses the design and implementation of PROVM and evaluates its effectiveness through

experiments on an ARM Cortex-M3 microcontroller. The results demonstrate that PROVM achieves predictable VM execution and provides a foundation for building secure and efficient virtualized systems on resource-constrained microcontrollers. The reference [8] examines the vulnerabilities of TrustZone technology on devices that lack memory protection mechanisms. The authors explore potential security vulnerabilities and attack vectors that can be exploited to compromise the integrity and confidentiality of data stored within the TrustZone environment. They provide an in-depth analysis of various attack scenarios, including buffer overflows, code injection, and privilege escalation techniques. The paper also discusses possible countermeasures to mitigate these vulnerabilities and enhance the security of TrustZone on devices without memory protection. The research serves as a valuable resource for understanding the security challenges associated with TrustZone technology and provides insights into potential attack vectors and defense strategies. There are many ways of obtaining IO virtualization on resource-constrained microcontrollers. As in [9], the authors propose a novel approach for achieving IO virtualization on resource-constrained microcontrollers. They present a lightweight virtualization framework that leverages the MPU to enforce isolation and provide secure and efficient access to IO resources for multiple tasks or applications. The paper discusses the design and implementation of the virtualization framework and evaluates its performance through experiments on an MPU-enabled microcontroller. The results demonstrate that the proposed approach effectively enables IO virtualization while incurring minimal overhead. The research contributes to the development of efficient virtualization techniques for microcontrollers, facilitating the deployment of complex and secure embedded systems. In [10], the authors address the challenge of providing memory protection across different hardware architectures without sacrificing performance. They propose an interface that abstracts the underlying memory protection mechanisms and provides a unified way of managing memory protection in Tock. The paper discusses the design principles and implementation details of the interface, highlighting its compatibility with various architectures. The authors evaluate the performance of the interface using benchmarking

experiments and demonstrate its effectiveness in achieving memory protection with low overhead. This research contributes to the advancement of operating system design by enabling architecture-agnostic memory protection in the context of the Tock operating system.

1.6 Objectives

This report's major goal is to give readers a thorough understanding of how the AXI protocol and MPU are verified in automotive microcontrollers, notably in the context of ADAS (Advanced Driver Assistance Systems). It highlights the value of AXI protocol verification to guarantee seamless communication between diverse parts and peripherals in automotive microcontrollers. To ensure dependable and compliant operation, this entails locating and fixing protocol breaches, timing problems, and interoperability problems. The research also aims to highlight the significance of MPU verification in enforcing memory access regulations, safeguarding sensitive data, and avoiding unauthorised access or corruption. The objective is to guarantee the integrity and security of the memory of the microcontroller by rigorously testing, assertion-based verification, and coverage analysis of the MPU tests. This research intends to highlight the crucial role of verification in automotive microcontrollers for ADAS applications by offering insights into the verification process and its significance.

1.7 Organization of Thesis

The rest of the thesis is laid out as follows:

Chapter 2: This chapter discusses how the AXI Protocol's Design Verification is carried out, how master-slave communication is formed, and the many forms of axi transactions.

Chapter 3: Design of Memory Protection Unit is discussed in this chapter. Furthermore, IP wrapper of MPU is developed by writing verilog code and waveform

results are seen.

Chapter 4: This chapter performs the verification of the memory protection unit. This involves the formal and functional testing of the MPU. Following the verification, the assertions and coverage results are analyzed to assess the effectiveness of the verification techniques employed.

Chapter 5: In this chapter, Conclusion includes the verification of the Memory Protection Unit (MPU) in automotive microcontrollers which is crucial for ensuring reliable and secure operation in Advanced Driver Assistance Systems (ADAS) applications.

Chapter 2

Design Verification of AXI Protocol

In the realm of System-on-Chip (SoC) design, seamless communication between different components plays a pivotal role in maximizing the overall performance of the system. To address this need, the Advanced eXtensible Interface (AXI) protocol, developed by ARM, has emerged as a widely embraced industry standard for interconnect protocols in SoC designs. This article explores the fundamental design principles, notable features, and advantages of the AXI protocol, highlighting its significance in enabling efficient communication and seamless integration within the system.

2.1 Overview of the AXI Protocol

The AXI protocol plays a pivotal role as a foundational framework for facilitating communication between IP blocks within a System-on-Chip (SoC). By offering a standardized and high-performance interface, it enables smooth and efficient data transfer and coordination among a wide range of components, including processors, memory controllers, and peripherals. Built on a master-slave architecture with a point-to-point connection, the AXI protocol supports concurrent data transfers, thereby promoting system scalability and adaptability.

Key design principles: The effectiveness of the AXI protocol in SoC communication can be attributed to several key design principles:

a. Burst-Mode Data Transfer: The AXI protocol supports burst-mode data transfers, allowing multiple data items to be transmitted in a single transaction. This feature significantly reduces transactional overhead and enhances data throughput by efficiently utilizing available bandwidth.

b. Separate Address and Data Channels: With the AXI protocol, the address and data channels are kept separate, enabling concurrent transactions and pipelining. This separation facilitates streamlined data streaming and reduces overall latency in the system, enhancing overall performance.

c. Out-of-Order Transaction Support: The AXI protocol incorporates out-of-order transaction execution, allowing the receiver to reorder transactions based on data availability. By optimizing resource utilization and minimizing idle time, this feature enhances system performance and efficiency.

d. Flexible Channels and Protocols: The AXI protocol offers a range of channel widths and protocols, including AXI4, AXI4-Lite, and AXI4-Stream, to accommodate diverse communication requirements. This flexibility empowers designers to customize the protocol to suit specific needs, ensuring optimal performance and resource allocation.

2.2 Design of AXI protocol

The AXI (Advanced eXtensible Interface) protocol, created by ARM, embodies a well-structured architecture tailored to enable streamlined and standardized communication among IP (Intellectual Property) blocks within a System-on-Chip (SoC) design. This protocol comprises several key components and channels, each fulfilling a specific role in facilitating efficient data exchange and coordination.

2.2.1 Master-slave communication:

The AXI protocol adopts a master-slave communication model, where a master device takes the lead in initiating transactions with one or more slave devices. Within this architecture, the master device assumes the responsibility of initiating read and write operations, while the slave devices promptly respond by either providing or receiving the requested data. This collaborative framework ensures efficient and synchronized data exchange between the master and slave components.

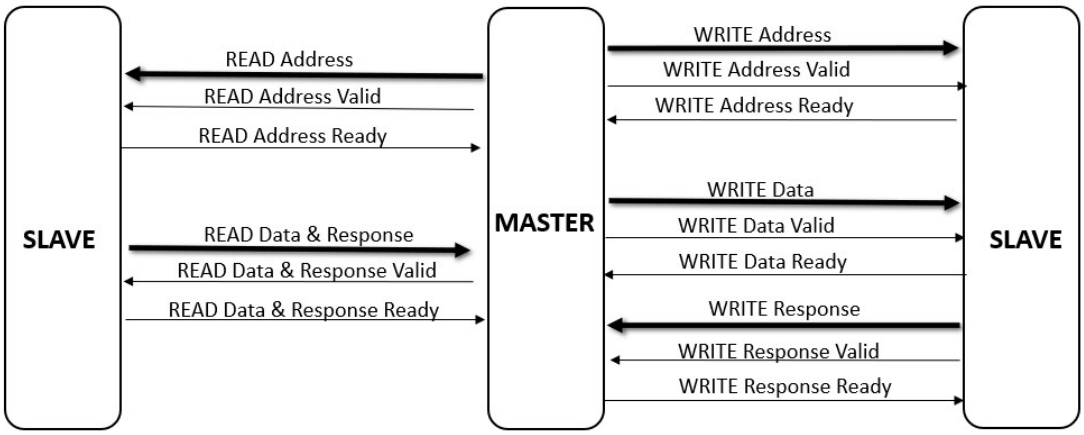

Figure 2.1: AXI Protocol Channels

Channels: The AXI protocol comprises different channels to facilitate the transmission of specific types of information between the master and slave devices. These channels include:

a. Address Channel:

Transmits address information from the master device to the slave device. Incorporates additional control signals, such as burst type, burst length, and transaction attributes.

b. Data Channels:

Read Data Channel: Transfers data from the slave device to the master device during read transactions [11].

Write Data Channel: Transfers data from the master device to the slave device during write transactions. Both data channels include a data bus for the actual data

transfer and associated control signals.

c. Response Channels:

Write Response Channel: Provides feedback from the slave device to the master device, indicating the status of write transactions.

Read Response Channel: Provides feedback from the slave device to the master device, indicating the status of read transactions. Transaction Types:

2.2.2 AXI transaction types:

The AXI protocol supports various transaction types to accommodate different communication requirements. These transaction types include:

a. Read Transactions: The master device initiates a data request from the slave device, indicating the specific information it requires. In response, the slave device promptly transfers the requested data through the designated read data channel. This bidirectional communication ensures seamless and reliable data exchange between the master and slave components, facilitating efficient operation within the system

b. Write Transactions: The master device transmits data to the slave device through the write data channel, providing the necessary information for further processing. The slave device receives the data, interpreting it accordingly, and performs the required actions based on the received information. This two-way communication between the master and slave components ensures effective data transfer and enables the slave device to execute the appropriate operations as instructed by the master device.

c. Exclusive Access Transactions: Ensure exclusive ownership of a designated memory location or resource, granting sole access to a single entity at a time. Support atomic operations or critical sections within the system, ensuring that these operations are executed as indivisible units without interference from concurrent processes or threads.

AXI Protocol Versions:

Over time, the AXI protocol has undergone several iterations to adapt to evolv-

ing design requirements. Prominent versions include AXI3, AXI4, and AXI4-Lite. Each iteration introduces enhancements and additional features while ensuring compatibility with previous versions.

The architecture of the AXI protocol, characterized by its master-slave communication model, separate channels for data and control signals, and support for diverse transaction types, fosters efficient and dependable communication between IP blocks in a System-on-Chip (SoC) design. This standardized architecture promotes seamless interoperability, scalability, and reusability, enabling the seamless integration of components from different vendors and facilitating the development of sophisticated and high-performance SoCs.

2.3 Verification of AXI Protocol

IP verification is the process of validating the functionality and correctness of an Intellectual Property (IP) block. It involves comprehensive testing and analysis to ensure that the IP meets its design specifications and performs as intended. There are two main approaches to IP verification:

Functional Verification:

Functional verification focuses on validating the functional behavior of the IP. Test cases are designed to cover different scenarios and use cases, thoroughly exercising the IP's functionality. By simulating real-world conditions, functional verification identifies and addresses any functional bugs or errors that may exist in the IP design.

Formal Verification:

Formal verification utilizes mathematical algorithms and logic reasoning to verify the correctness of the IP design. It involves creating formal models and properties, and then using formal verification tools to mathematically analyze the design for adherence to these properties. By exhaustively exploring all possible scenarios, formal verification helps ensure that the IP design is free from logical errors and satisfies the specified requirements.

Both functional and formal verification approaches are crucial in IP verification, complementing each other to provide comprehensive validation. Functional verification validates the IP's behavior in practical scenarios, while formal verification provides mathematical proofs or counterexamples to verify the design's correctness. Together, these approaches enhance the reliability and quality of the IP.

Here for Verification of AXI Protocol, Functional verification is done using UVM Methodology.

Snippet of UVM Code for AXI Protocol Verification

```
int main (void) {
INFO(STIMNAME, " Stimulus Starts");
write32(REG32(0*60000044),0*12340000);
read_expected32(REG32(0*60000044),0*12340000);

INFO(STINAME, " Stimulus Ends");
return 0;
}
```

Listing 2.1: Snippet of UVM Code for AXI Protocol Verification

Below is shown the waveform results for the same:

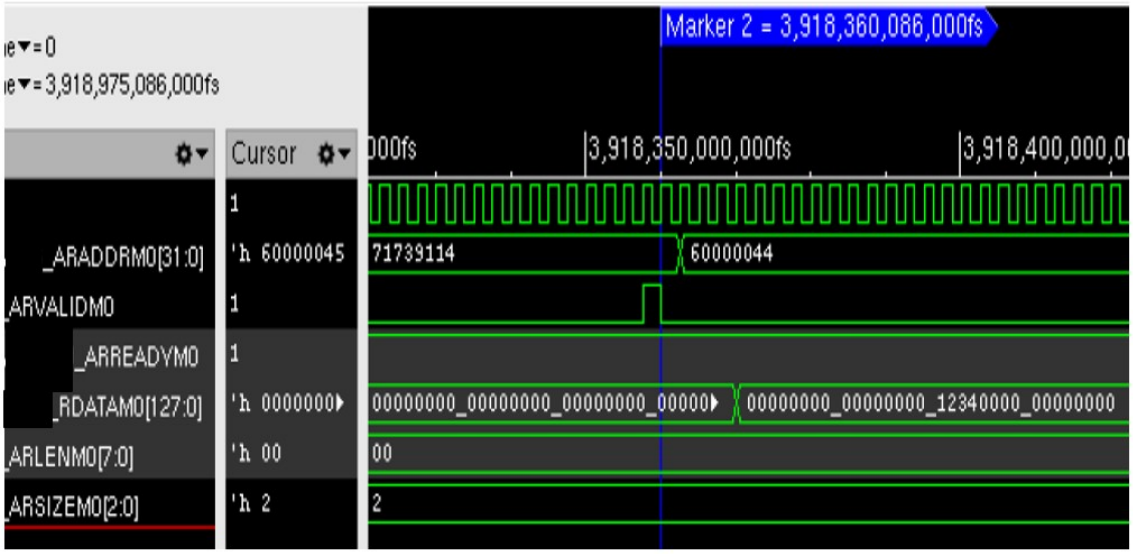

Figure 2.2: AXI Functional Verification Waveform

Description: In the given scenario, the Master is attempting to write to a memory location in the SRAM with a starting address of 32'h60000044. Before sending this address, a handshaking process is carried out between the Master and the memory interface.

The handshaking process involves two control signals: ARVALID and ARREADY. ARVALID indicates that the Master is ready to send the address, while ARREADY signifies that the memory interface is ready to receive the address. Both signals need to be high simultaneously to ensure proper handshaking [12].

Once the handshaking is completed successfully, the Master sends the address 32'h60000044 to the memory interface, indicating the desired memory location for the write operation.

Similarly, a handshaking process is performed before retrieving the RDATA (read data) from the slave (SRAM). This handshaking process ensures that both the Master and the memory interface are ready to transfer data.

Control fields other than the address, such as length and size, are used to determine the amount and format of data to be read from the specific memory location. These control fields provide information about the length of the data transfer and the size of each data element. This facilitates data transfer and ensures that the correct data is read from or written to the intended memory location within the SRAM.

2.4 Summary

AXI (Advanced eXtensible Interface) design verification involves making sure the design complies with the specifications and requirements stated in the AXI standard. It entails examining the AXI interface's proper timing, functionality, and protocol compliance within the architecture. The AXI protocol's design verification procedure is summarised as follows:

Functional Verification: In this phase, it is ensured that the design operates as expected and functions in accordance with the requirements of the AXI protocol.

Various scenarios, boundary cases, and corner cases are tested to make sure the design appropriately addresses all potential input circumstances.

Protocol Conformity: The design is examined in accordance with the particular version of the AXI protocol it implements, such as AXI3 or AXI4. By doing this verification, you can be sure that the design complies with the AXI protocol's established guidelines, signalling norms, and transaction ordering specifications.

Performance Verification: The performance of the design is assessed to make sure it satisfies the expected throughput and latency criteria of the AXI protocol. Data transfer rates, response times, and other metrics are evaluated as part of performance testing to make sure the design performs within the expected performance constraints.

Verification Environment: Testbenches, stimulus generation, and coverage analysis are all included in the creation of a thorough verification environment. In order to identify and fix possible problems or faults, this environment is used to replicate and validate the behaviour of the design.

In order to assure the design's accuracy, compliance, timeliness, and performance, the AXI protocol design verification procedure is demanding. The design is proven to meet the criteria and specifications of the AXI protocol through detailed functional testing, protocol compliance tests, timing verification, and a full verification environment.

Chapter 3

Design of Memory Protection Unit

3.1 Introduction to Memory Protection Unit

In the Network-on-Chip (NoC) architecture of an automotive microcontroller, the Memory Protection Unit (MPU) is very important for memory protection and access control. The MPU is in charge of enforcing stringent rules surrounding memory access throughout the system thanks to its integrated hardware design. It administers and regulates the access rights granted to various NoC-connected components, also referred to as Masters.

The MPU performs the following crucial tasks:

Memory Access Control: The MPU determines which Masters are permitted to read from or write to particular memory locations by defining and enforcing access rights. This makes sure that each component can only access the places that are allowed by the rules that have been created.

Address Range Validation: By comparing them to established address ranges, the MPU determines whether memory addresses requested by Masters are valid. With the help of this validation procedure, the MPU can limit access depending on the designated memory regions.

Error Management: The MPU recognises and manages incorrect memory access attempts, such as those that violate access restrictions or access prohibited areas. Automotive microcontrollers create a safe and controlled environment for

memory access by integrating an MPU into the NoC. This contributes to increased safety in automotive applications by preventing unauthorised access, protecting against data corruption, and maintaining the integrity and reliability of the system. The MPU is in charge of enforcing memory protection and access permissions, making sure that various memory areas are accessed in accordance with established guidelines and limitations.

The MPU Design that is being discussed is a particular Memory Protection Unit implementation or design. It consists of different blocks and logic that carry out checks and validations to make sure that memory accesses follow the specified rules and configurations. These checks include checking for border checks, multi-region faults, access types, and VirtualID matching. With its check blocks, error generation logic, and interface signals, the MPU Design attempts to improve system security and safeguard memory by enforcing access limits and spotting any violations or errors in memory accesses. To summarise, the MPU in the context in which it is used stands for a Memory Protection Unit, a hardware element in charge of maintaining memory access control and security in a system.

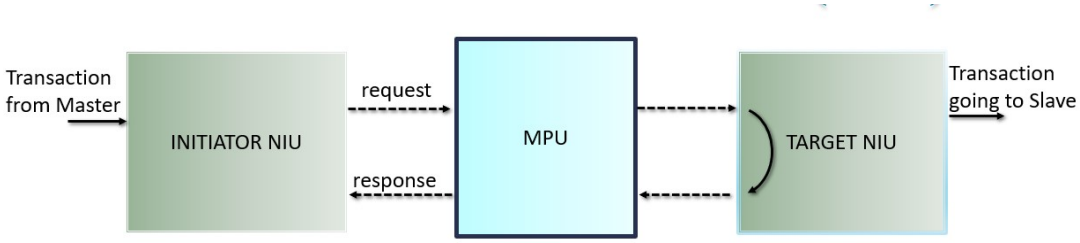

Figure 3.1: Memory Protection Unit

3.2 Block Design of MPU

The system in the MPU Design, as shown in Figure 3.2, is made up of a number of interconnected blocks that cooperate to carry out checks and produce mistakes. The APB interface is used to configure the MPU registers at the beginning of the design process. These registers store data related to different parts of the system,

including read/write accesses, VirtualID accesses, MasterID accesses, the type of each region in the SRAM, the memory type, and read/write accesses. The system receives input via the AXI interface, which is subsequently sent to the MPU check blocks after the MPU registers have been configured.

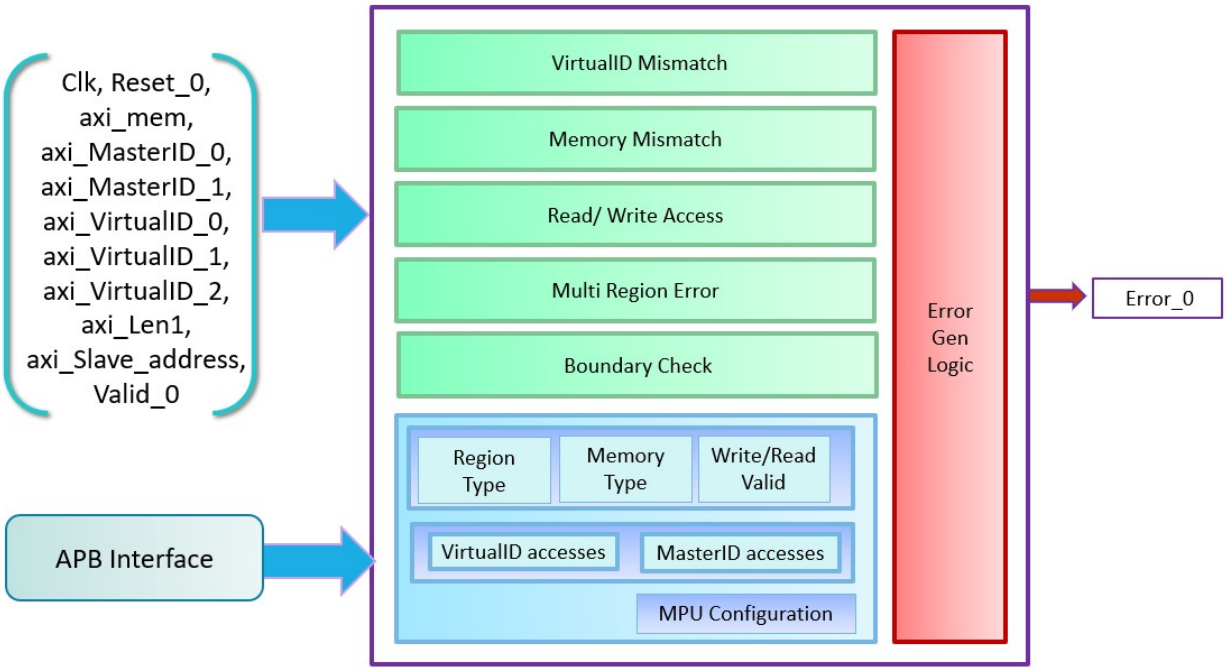

Figure 3.2: MPU Block Design

These check blocks are in charge of carrying out the following five different checks:

Virtual Machine Matched or Mismatched: This check evaluates whether or not the input's VirtualID matches the relevant region's VirtualID by comparing the two.

Memory Matched or Mismatched: This check determines whether the input's memory type corresponds to the anticipated memory type set up for the region.

Access Type (Read/Write/Execute): This check checks the input's intended access type (read, write, or execute) and verifies it using the access rights that have been set up.

Multi-Region Error: This check keeps track of how many regions are accessed and checks to see if many regions are being accessed at once, which may be an error.

Boundary Check: By making sure the address of the data access does not cross the region's border, this check guards against unauthorised access.

The error generation logic block is then supplied with the results of these checks. The error generating logic determines whether an error condition exists and produces the appropriate error output based on the findings of the tests. The MPU registers are configured, inputs are routed through check blocks, and the results are examined by the error generating logic to produce error outputs. Overall, this MPU Design follows a systematic sequence.

If Output Error_0 is 1: means MPU will stop Master from accessing Slave.

If Output Error_0 is 0: means MPU Check was passed and there is successful transaction between Master and Slave.

3.2.1 Snippet of Verilog Code for MPU Design

```
assign VirtualID_ok           =  ((region_virtualID
                                  [index_region]
                              == pkt_vmid) && pkt_vld) ?
                                  1'b1 : 1'b0 ;
assign VirtualID_matched      = vmid_ok & pkt_vld1;
assign mem_mismatched         = mem_mismatch & pkt_vld ;
assign VirtualID_not_matched  = (~VirtualID _ok)   &
                                    pkt_vld1;
assign multi_region_error     = (( region_hits > 6'd1) ?
                                  1'b1 : 1'b0 ) ;
assign boundry_nok            = (|bound_cross);
assign Error_intermediate     = (VirtualID_not_matched |
                                  mem_mismatched |
                                  multi_region_error |
                                  read_write_access |
                                  boundry_nok) ;
always @ (posedge Clk) begin
    if ( pkt_vld2 )
      Error_0 = Error_intermediate ;
end
```

Listing 3.1: Snippet of Verilog Code for MPU Design

Description:

The code first calculates individual checks or conditions (VirtualID_not_matched, mem_mismatched, multi_region_error, read_write_access, boundry_nok) and then combines their results using a logical OR operation. The combined result is assigned to the variable Error_intermediate. This variable represents the intermediate error condition based on the combination of different checks. By combining these individual checks, the Error_intermediate variable captures any error condition that occurs in any of the checks. If any of the individual checks evaluate to true, indicating an error, then Error_intermediate will also be true. Otherwise, if all the individual checks evaluate to false, Error_intermediate will be false. The Error_intermediate value is then used in the subsequent logic or further processing, such as assigning it to another variable (Error_0 in this case) based on certain conditions or events (in this case, when pkt_vld2 is true and triggered by the positive edge of the Clk signal).

3.2.2 IP Wrapper of MPU

Upon generating the IP (Intellectual Property) wrapper for the Verilog code of the MPU Design, an interface is provided to facilitate communication with the IP. This interface includes the necessary AXI (Advanced eXtensible Interface) signals as inputs to enable the execution of the checks within the IP [13].

The IP and the surrounding systems can communicate with one another through the AXI interface signals. These signals enable the system to give the IP inputs needed to carry out check procedures. Address signals, data signals, control signals, and status signals are among the unique AXI signals needed for the MPU design. The Advanced Peripheral Bus (APB) interface signals are included as inputs in the MPU design in addition to the AXI interface signals. The setting of registers inside the MPU block is made possible by these APB interface signals. The APB interface signals are used for tasks including defining read/write accesses, VirtualID accesses, and MasterID accesses, as well as setting the type of each area in SRAM and designating the memory type. ese configurations are necessary to modify the

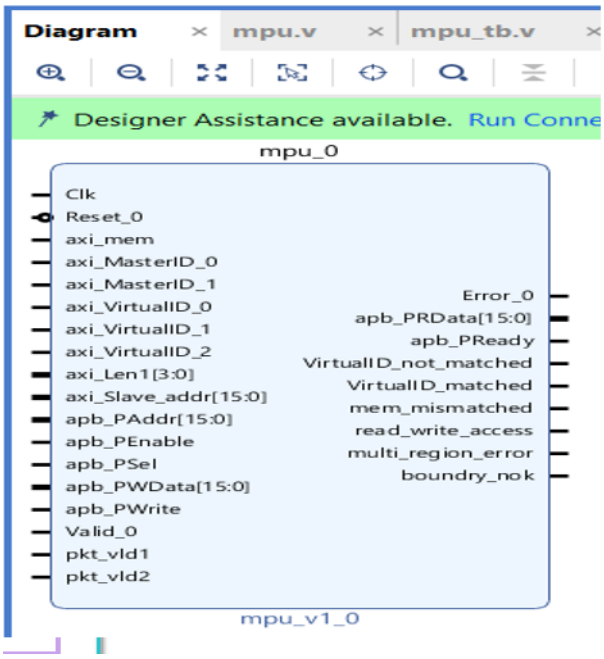

Figure 3.3: IP wrapper of MPU

MPU Block's behaviour and attributes in accordance with system requirements [14].

The IP wrapper enables the system to effectively communicate with and use the MPU Design by using the AXI interface signals as inputs. It enables the system to give the required inputs for the checks to be performed by the IP, simplifying the verification of virtual machine matching, memory consistency, access types, multi-region faults, and boundary checks as stated within the MPU Design.

3.2.3 Waveform results of MPU design

In Waveform 3.4, the behavior of the MPU Design is illustrated. Specifically, when the check for VirtualID mismatch occurs, the Error_0 output is observed to transition to a high state.

The timing and interactions of the signals inside the design during a simulation or testing scenario are captured by the waveform. It aids in signal transmission and system response visualisation.

When the VirtualID check determines a mismatch between the input VirtualID and the corresponding region's VirtualID, it triggers an error condition. As a result,

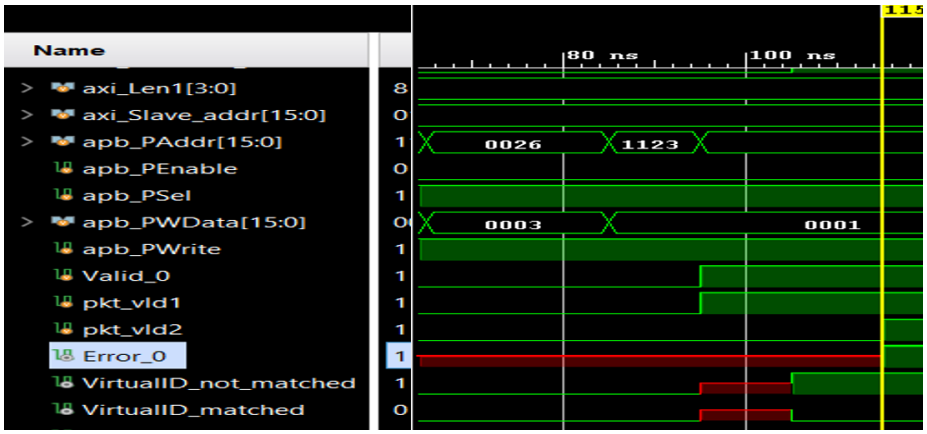

Figure 3.4: Waveform results of MPU Design

the Error_0 output signal undergoes a transition, indicating the presence of an error. Waveform 3.4 provides a clear representation of the response of the MPU Design when encountering a VirtualID mismatch, highlighting the generation of the error output, Error_0.

3.3 Summary

The MPU Design consists of multiple blocks and functionalities aimed at performing various checks and generating error outputs. Here is a summary of the design:

Configuration: The design begins with the configuration of MPU registers through the APB interface. These registers store information about region types, memory types, access types, VirtualID accesses, and MasterID accesses.

Check Blocks: The AXI interface receives inputs after register configuration, which are then handled by the MPU check blocks. These blocks run five different types of checks:

a. Virtual Machine Matched or Mismatched: Checks to see if the input VirtualID matches the VirtualID for the associated area.

b. Memory Matched or Mismatched: Checks to see if the input's memory type corresponds to the region's set memory type.

c. Access Type (Read/Write/Execute): This check compares the desired access type to the access permissions that have been set up.

d. Multi-Region Error: Tracks how many regions are accessed and recognises when several regions are accessed at once.

e. Boundary Check: This step confirms that the data access address does not go beyond the region's border, preventing unauthorised access.

Error Generation: The outputs of the check blocks are fed into the error generation logic. Based on the results of the checks, this logic determines if an error condition exists and generates an appropriate error output.

IP Wrapper: Finally, an IP wrapper is generated to encapsulate the Verilog code of the MPU Design. It provides an interface, including AXI signals, to allow external systems to communicate with the IP and provide inputs necessary for the checks to be performed.

In summary, the MPU Design includes configuration registers, check blocks for different types of checks, error generation logic, and an IP wrapper with an AXI interface. Together, these components enable the design to verify virtual machine matching, memory consistency, access types, multi-region errors, and boundary checks, generating error outputs based on the check results.

Chapter 4

Verification of Memory Protection Unit

Verification is the process of determining and validating whether a design or system is accurate, functional, and compliant with its needs or specifications. In order to make sure that the design fulfils the specified functionality and acts as predicted, a variety of techniques, processes, and tools are used. Before the system is deployed or manufactured, verification tries to find and fix design flaws, defects, and functional problems.

Functional verification and formal verification are the two main types of verification that are frequently employed in the industry.

Functional Verification: The goal of functional verification is to confirm that a design is functionally sound. To replicate and evaluate the design under various operating situations, test scenarios, test cases, and test benches must be created. The objective is to guarantee that the design executes its intended functions accurately and as intended.

Formal Verification: To ensure that a design is right, formal verification uses formal procedures and mathematical reasoning. It entails building a mathematical model of the design and thoroughly examining its behaviour and properties using formal techniques like model checking, theorem proving, and equivalence checking.

A design's accuracy and quality can be ensured through both functional and

formal verification. While formal verification offers mathematical rigour to verify certain features or investigate all potential design states, functional verification focuses on proving the design's intended utility. confidence in the accuracy and dependability of the design can be increased by combining the two approaches.

4.1 Introduction to Verification of MPU

An essential stage in ensuring the correct and dependable operation of a Memory Protection Unit (MPU) in implementing memory protection and access control is the verification of the MPU. Validating the MPU design's performance, compliance, and functionality is part of the verification process. Before the MPU is integrated into the larger system, potential problems and faults can be found and fixed by carefully testing the MPU against the defined requirements and confirming its behaviour in various circumstances.

With reference to Figure 4.1, **Four-phase verification cycle:**

Development: A number of tasks are carried out during the development phase to lay the groundwork for the verification process. The design of the MPU's architecture includes a description of the memory regions, access control procedures, and other pertinent characteristics. The development of test benches and sequences creates a simulation environment for delivering test stimuli and confirming the behaviour of the MPU.

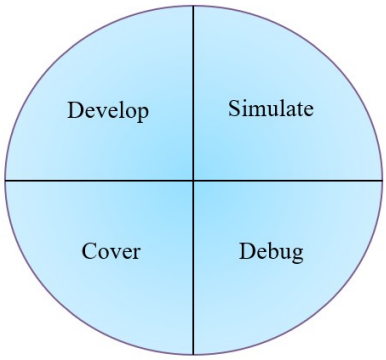

Figure 4.1: Verification Phases

Simulation: Using the appropriate simulation tools, the generated testbenches and sequences are executed during the simulation phase. To generate the simulation model, the verification environment—which includes the testbench and sequences—is compiled and elaborated. Waveforms are produced during simulation to simulate the behaviour of the MPU and other system parts. The simulation approach aids in validating the MPU's behaviour and operations under various test situations.

Debugging: The goal of the debugging phase is to find and fix problems that arise during simulation. Investigating any anomalies, mistakes, or unusual behaviour seen in the simulation waveforms is part of this process. To identify the source of problems, debugging operations may require looking into transactional or signal-level details. Design flaws, false assumptions, or problems in the MPU or testbench can be found and addressed by diligent study and troubleshooting.

Coverage: Assessment of the efficiency and thoroughness of the verification process is done during the coverage phase. To measure various aspects of verification, several coverage metrics are used. Functional coverage measures how effectively the verification tests use the MPU's various features to test all planned behaviours. Code coverage gauges how thoroughly the verification tests have put the RTL code to the test. To evaluate the coverage of particular attributes or assertions defined in the design, SystemVerilog Assertions (SVA) coverage can be used. The first step is to receive input on the coverage findings, which is used to improve the verification plans and methods moving forward.

The development, simulation, debugging, and coverage steps make up the iterative verification cycle. The verification process is improved upon and repeated when problems are found and fixed until the MPU design complies with the required requirements, conforms to the intended behaviour, and satisfies the desired coverage targets.

4.2 Functional Verification of MPU Checks

The MPU check blocks' ability to perform the five separate tests on the input received through the AXI interface is confirmed through functional testing. Creating test cases that cover numerous scenarios for each check, properly establishing the MPU registers, and running the tests are all steps in the verification process. The checks are verified by matching the input's VirtualID with the VirtualID of the pertinent region, confirming the access type, keeping track of multi-region accesses, and applying boundary checks to prevent unauthorised access. Functional testing guarantees that the MPU check blocks effectively enforce the expected behaviour and offer trustworthy protection for memory accesses [15].

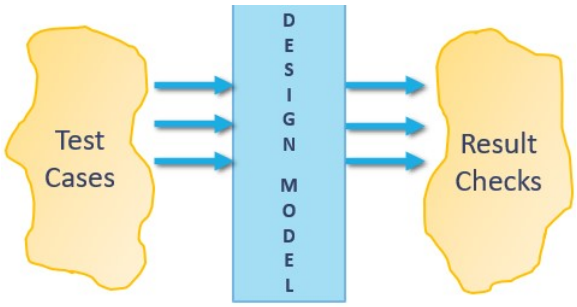

Figure 4.2: Functional Verification

4.2.1 Virtual Machine matched or mismatched check

Virtualization is a term used to describe the ability of hardware to simultaneously run several operating systems (OS). With its assistance, one can construct virtual machines (VMs), which are independent instances of an OS running on a single physical computer. A hypervisor, is a software that is responsible for creating and managing VMs. The hypervisor controls how the virtual machines are allocated CPU, memory, storage, and network resources. A hypervisor is a boss software that decides which type of permission the OS have on the particular resource [16].

The "Virtual Machine Matched or Mismatched" check is an essential part of the verification process in the context of virtualization. It evaluates whether the

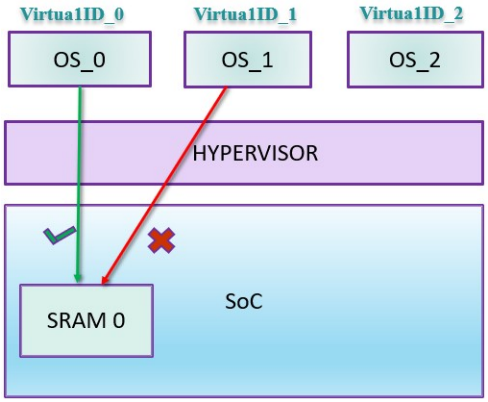

Figure 4.3: SoC interaction with Operating Systems

VirtualID assigned to an input matches the relevant region's VirtualID, thereby determining if the virtual machine accessing the resource is authorized. The idea of virtualization is the creation and operation of numerous virtual computers (VMs) on a single physical hardware. A VirtualID, a special identity given to each Operating System, is used to isolate and distinguish one VM from another. The hypervisor, which serves as the boss programme, is in charge of setting up and overseeing these virtual computers.

The system checks to see if the VirtualID associated with the incoming request matches the VirtualID associated with the pertinent area or resource during the "Virtual Machine Matched or Mismatched" check. If there is a match, it signifies that the requesting VM is authorized to access the particular resource. However, if there is a mismatch, it implies that the VM does not have the necessary permissions to access the resource, and the request is denied.

This check ensures that each VM operates within its designated boundaries and prevents unauthorized access to resources. It contributes to maintaining isolation and security in a virtualized environment by enforcing access control and preventing interference between different virtual machines.

Overall, virtualization allows for the simultaneous execution of multiple operating systems or applications on a single physical machine. Through the use of VirtualIDs and associated checks like "Virtual Machine Matched or Mismatched,"

virtualization ensures that each virtual machine operates securely within its designated boundaries. Each OS is assigned a distinct VirtualID by the hypervisor according to the system architecture.

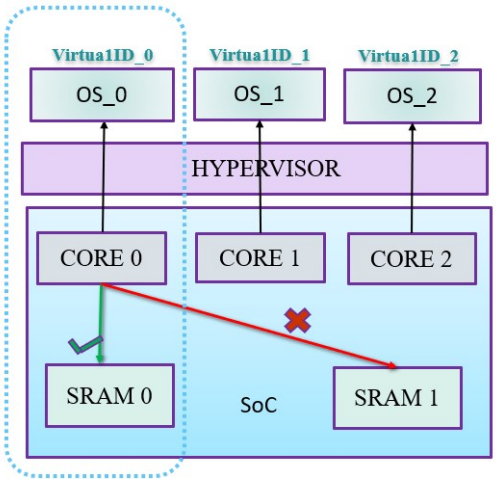

Figure 4.4: Virtual Machine Matched and Mismatched

With reference to Figure 4.4, VirtualID_0 is given to OS_0 and then assigned to CORE_0. The MPU registers are configured through the APB interface to manage access rights. In particular, access to SRAM_0 by CORE_0 with VirtualID_0 is permitted by the MPU register setup, whereas access to SRAM_1 by CORE_0 with VirtualID_0 is limited.

Snippet of UVM code for Virtual Machine matched or mismatched:

```
class VM_match_mismatch_test extends mpu_test;
 VM_match_mismatch_seq_h VM_match_mismatch_seq_t;
 `uvm_component_utils(VM_match_mismatch_test)

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

 VM_match_mismatch_pkt_seq_t::get_type());
 VM_match_mismatch_seq_t= VM_match_mismatch_seq_h::type_id::create(
   "       VM_match_mismatch_seq_t");
```

```
11  task run_phase(uvm_phase phase);
12      VM_match_mismatch_seq_t.mpu_base_class = mpu_base_class;
13      VM_match_mismatch_seq_t.start(dut_top_env_t.vs);
14  endtask: run_phase
15 endclass: VM_match_mismatch_test
```

Listing 4.1: Snippet of UVM Code for Virtual Machine Matched or Mismatched

Description: The MPU permits access when Master CORE_0 tries to access an area in Slave SRAM_0 in accordance with the setup. The MPU approves the operation since CORE_0 has been given VirtualID_0, which is permitted access to SRAM_0. By comparing the VirtualID of the requesting entity with the configured permissions, the MPU conducts a verification and decides as a result. However, the MPU blocks Master CORE_0 from accessing a region of Slave SRAM_1 when it attempts. The MPU register setting expressly states that CORE_0, associated with VirtualID_0, does not have permission to access SRAM_1 as the cause of the denial. In order to avoid any unauthorized access or potential security flaws, the MPU carefully enforces the configured limitations.

Virtual Machine matched waveform:

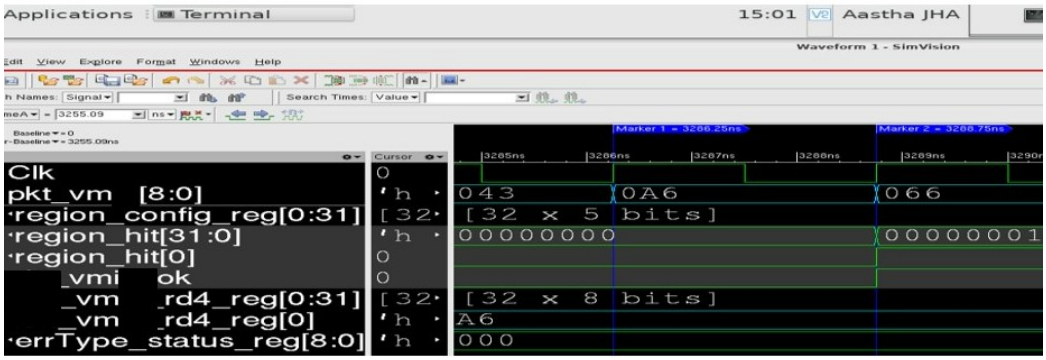

Figure 4.5: Virtual Machine Matched Waveform

A specific area of the SRAM in the provided waveform 4.5, is set up to only permit access to VirtualID = 8'hA6, as specified in the MPU (Memory Protection Unit) register. A "Virtual Machine matched" scenario is confirmed when the waveform shows that the VirtualID 8'hA6 is attempting to access the SRAM region.

This indicates that the expected VirtualID that has been set up for the SRAM area and the VirtualID connected with the accessing entity are same.

The access is therefore regarded as legitimate and authorised. The VirtualID 8'hA6 can easily access the SRAM region thanks to the technology. This matching VirtualID makes ensuring that the operating system or virtual machine that is accessing the resources in the designated SRAM area has the required authorizations.

Virtual Machine mismatched waveform:

Figure 4.6: Virtual Machine Mismatched Waveform

A specific area of the SRAM in the provided waveform 4.6 is set up to only permit access to VirtualID = 8'hA6, as specified in the MPU (Memory Protection Unit) register. The waveform, however, suggests a "Virtual Machine mismatched" scenario when VirtualID 8'h9B is trying to access the SRAM area. This indicates that the intended VirtualID (8'hA6) defined for the SRAM region does not match the VirtualID (8'h9B) associated with the accessing entity. The access is therefore seen as being illegitimate and unauthorised. The SRAM area is not accessible to the VirtualID 8'h9B by the system. This mismatched VirtualID shows that the operating system or virtual machine that is accessing the resources within the specified SRAM area does not have the requisite permissions.

In conclusion, this waveform shows an example of virtual machine mismatch, where the accessing entity's virtual ID (8'h9B) differs from the virtual ID (8'hA6) that is defined in the MPU register for the SRAM area. This helps to maintain the security and integrity of the virtualized system by ensuring that attempts at unauthorised access are thwarted.

4.2.2 Memory-type matched Or mismatched check

Memories can be broadly classified into two types:

1. **Normal Memory**
2. **Device memory**

Normal memory includes RAM, ROM, NVM, and Flash memories, while device memory refers to the registers present in cores. Additionally, memories can have different characteristics such as cacheable, non-cacheable, bufferable, and non-bufferable, which determine their behavior and accessibility.

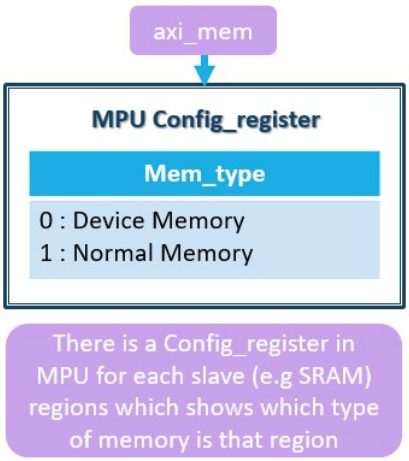

Figure 4.7: Memory-type Matched Or Mismatched

In the context of Memory-type matched Or mismatched checks, the first step involves configuring the MPU Config_register for each slave (SRAM) region. With reference to Figure 4.7, configuration specifies the type of memory associated with that particular region. When the master initiates an access by sending the axi_mem signal, it indicates the desired memory type that the master wants to access. The MPU checks this axi_mem signal and compares it with the memory type written in its Config_register for the corresponding slave region. If there is a match between the axi_mem signal and the configured memory type, it signifies a "Memory type Match" case, indicating that the desired memory type matches the configured type. In this case, the check is considered successful (pass), and the MPU allows the transaction to proceed to the slave for accessing the memory region.

On the other hand, if there is a mismatch between the axi_mem signal and the configured memory type, it represents a "Memory type Mismatch" case. This indicates that the desired memory type does not match the configured type in the MPU Config_register. As a result, the check fails, and the MPU prevents the transaction from progressing to the slave, effectively halting the transaction. The Memory type match/mismatched check ensures that the desired memory type specified by the master aligns with the configured memory type in the MPU Config_register. This validation helps in maintaining consistency and preventing unauthorized or incompatible memory accesses within the system.

Snippet of UVM Code for Memory-type matched Or mismatched:

```
class Memtype_match_mismatch_test extends mpu_test;
 Memtype_match_mismatch_seq_h Memtype_match_mismatch_seq_t;
 `uvm_component_utils(Memtype_match_mismatch_test)

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

 Memtype_match_mismatch_pkt_seq_t::get_type());
 Memtype_match_mismatch_seq_t= Memtype_match_mismatch_seq_h::
   type_id::create("       Memtype_match_mismatch_seq_t");
 task run_phase(uvm_phase phase);
    Memtype_match_mismatch_seq_t.mpu_base_class = mpu_base_class;
    Memtype_match_mismatch_seq_t.start(dut_top_env_t.vs);
 endtask: run_phase
endclass: Memtype_match_mismatch_test
```

Listing 4.2: Snippet of UVM Code for Memory-type Matched Or Mismatched

Description:

In the provided waveform 4.8, there is a specific signal called mem_mismatched that goes high at a certain point in time. This signal indicates a memory type

mismatch, suggesting that the desired memory type requested by the master does not match the configured memory type in the MPU.

When the mem_mismatched signal goes high, it triggers the error generation logic within the MPU. The error generation logic detects the mismatch and identifies it as an error condition. In the subsequent clock cycle, the Error_0 signal is raised, indicating the occurrence of an error.

The rising of the Error_0 signal serves as a notification to the system or external entities that a memory type mismatch error has occurred. This signal can be captured by error handling mechanisms or used to trigger specific actions within the system, such as error logging, interrupt generation, or system reset.

Memory-type mismatched waveform:

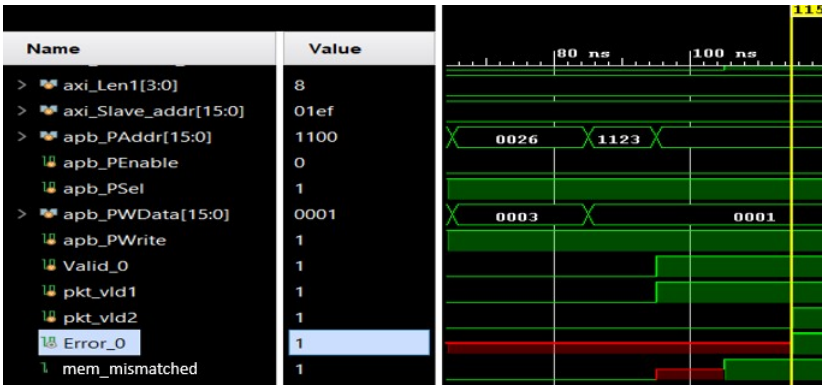

Figure 4.8: Memory-type Matched Or Mismatched

By observing the waveform 4.8 and the sequence of events, it becomes evident that the mem_mismatched signal acts as a trigger for error detection, while the Error_0 signal reflects the occurrence of a memory type mismatch error. These signals play a crucial role in the verification process of the MPU, allowing for the identification and handling of memory type mismatch scenarios to ensure the integrity and reliability of memory access within the system.

4.2.3 Access type (Read/Write/Execute) check

In the given scenario, the MPU (Memory Protection Unit) contains separate registers for each slave (SRAM) region, and each register is associated with a specific

MasterID. These registers define the permissions granted to each master for accessing the corresponding SRAM region.

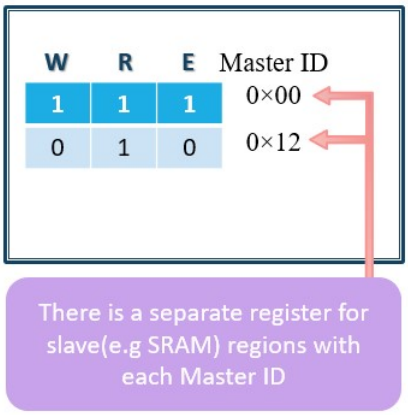

Figure 4.9: Access Type (Read/Write/Execute)

For instance, with reference to Figure 4.9 the MasterID 8'h00, which corresponds to CORE_0 master, has permissions to write, read, and execute in a particular region of the SRAM (region number 29) as specified in the MPU register. This means that CORE_0 is authorized to perform read, write, and execute operations in region 29 of the SRAM.

On the other hand, the MasterID 8'h12, corresponding to CORE_1 master, is granted only read permission in the same region (region 29) of the SRAM. CORE_1 is allowed to read data from region 29, but it does not have the permission to write to or execute code from that region.

Now, if CORE_1 attempts to write to the memory region 29 of the SRAM, it will encounter an error. The MPU performs checks based on the MasterID and the desired operation. Since CORE_1 does not have write permission for region 29, the MPU detects a violation and terminates the transaction, preventing CORE_1 from performing the unauthorized write operation.

However, when CORE_0 tries to write to memory region 29 of the SRAM, the MPU recognizes that CORE_0 has the necessary write permission for that region based on its MasterID. As a result, the MPU allows the transaction to proceed, and the write operation is successfully executed in the designated memory region.

Snippet of UVM Code for Access type (Read/Write/Execute):

```
class write_access_error_test  extends mpu_test ;
   write_access_error_packet_t;
   write_access_error_seq_t write_access_error_seq_h;
  `uvm_component_utils(write_access_error_test)

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

   virtual function void build_phase(uvm_phase phase);
   write_access_error_seq_h = write_access_error_seq_t::type_id::
   create("   write_access_error_seq_h");

  super.build_phase(phase);
  endfunction : build_phase
  virtual task run_phase(uvm_phase phase);
     write_access_error_seq_h.mpu_base_class = mpu_base_class;
  endtask : run_phase
endclass: write_access_error_test
```

Listing 4.3: Access Type (Read/Write/Execute)

Description:

In summary, the MPU's separate registers for each slave region, along with specific permissions assigned to different MasterIDs, enable controlled and secure access to memory resources. The MPU verifies the permissions associated with the MasterID and the requested operation, ensuring that unauthorized accesses are detected and prevented, thus maintaining the integrity and security of the system. A write operation to the 29th area of the SRAM, started by Master CORE_1, can be seen in the waveform 4.10. To verify the access privileges for that particular area and MasterID combination, the MPU registers in place carry out a permission check.

Access Type (Read/Write/Execute) waveform:

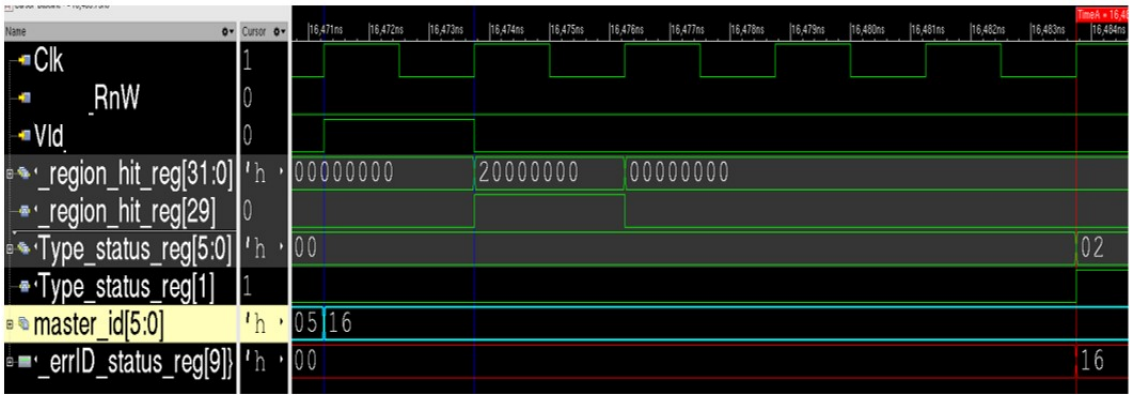

Figure 4.10: Access Type (Read/Write/Execute)

The MPU finds that CORE_1 does not have the necessary permissions to write to the specified region after performing the permission check. The MPU consequently recognises this as an incorrect scenario.

The MPU generates an error ID in response to the error detection, letting CORE_1 know that the access attempt was not authorised. The error ID is used as a point of reference to find the specific MasterID connected to the violation.

The master who allowed unapproved access is effectively highlighted by the MPU by reporting the error ID, assisting in problem identification and resolution. This error reporting mechanism plays a crucial role in preserving the system's integrity and security by quickly identifying and stopping unlawful entry attempts.

4.2.4 Multiple regions access check

The Multi-Region Error check in the MPU ensures that at any given time, a master can only access a single region of the SRAM. If multiple regions are accessed simultaneously by the same master, it is considered a violation of this rule and is flagged as a multi-region hit error.

In the provided scenario, Figure 4.11, when CORE_0 attempts to access more than one region of the SRAM simultaneously, the MPU detects this violation. As a result, the MPU triggers the multi-region hit error, indicating that CORE_0 has ac-

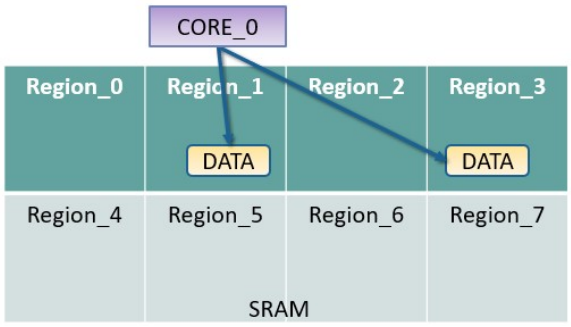

Figure 4.11: Multiple Regions Access

cessed multiple regions concurrently. To rectify the situation, the MPU takes action to terminate both transactions initiated by CORE_0 that involve accessing multiple SRAM regions simultaneously. By terminating these transactions, the MPU ensures compliance with the rule of allowing access to only a single region at a time.

Snippet of UVM Code for multiple regions access:

```
class multi_access_error_test  extends mpu_test ;
   multi_access_error_packet_t;
   multi_access_error_seq_t multi_access_error_seq_h;
  `uvm_component_utils(multi_access_error_test)
  function new(string name = " multi_access_error_test",
   uvm_component parent);
      super.new(name,parent);
  endfunction : new

   virtual function void build_phase(uvm_phase phase);
   multi_access_error_seq_h = multi_access_error_seq_t::type_id::
   create("   multi_access_error_seq_h");
   endfunction : build_phase
  virtual task run_phase(uvm_phase phase);
     multi_access_error_seq_h.mpu_base_class = mpu_base_class;
  endtask : run_phase
endclass: multi_access_error_test
```

Listing 4.4: Multiple Regions Access

Description:

The purpose of the Multi-Region Error check is to enforce proper access behavior and prevent potential conflicts or inconsistencies that may arise from accessing multiple regions simultaneously. By identifying and addressing such violations, the MPU contributes to maintaining the integrity and reliability of the system's memory access operations

In the provided waveform 4.12, it can be observed that Master CORE_0 attempts to access two different slave addresses simultaneously. The slave addresses in question are 32'h60000040 and 32'h60000500.

Multiple Regions access waveform:

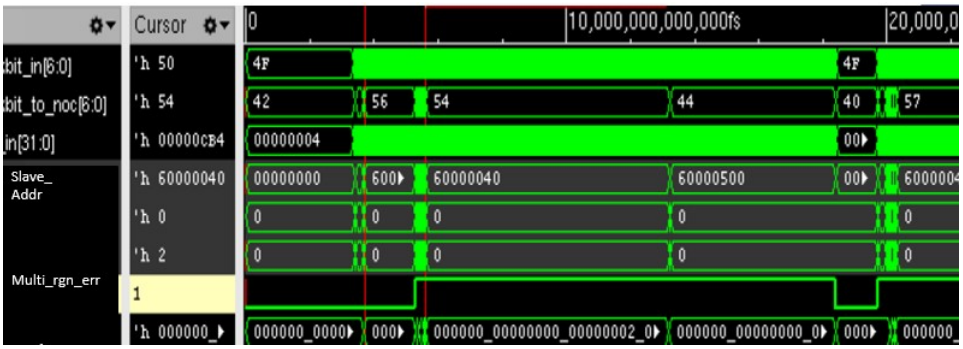

Figure 4.12: Multiple Regions Access

However, the MPU has a rule that allows a master to access only one region at a time. When CORE_0 tries to access both slave addresses simultaneously, it violates this rule, resulting in a multi-region error.

Upon detecting the multi-region error, the MPU takes action to halt the ongoing transaction. It stops the transaction from proceeding further to ensure compliance with the rule of accessing only a single region at a time.

The MPU ceases the transaction to avoid any potential conflicts or inconsistencies that might result from simultaneously accessing several areas. This provides correct coordination between the master and slave components as well as the reliability of the system's memory access operations.

The multi-region error detection and transaction termination tasks carried out by the MPU support effective memory management and thwart illegal or conflicting

access attempts. These actions add to the system's overall safety and dependability.

4.2.5 Boundary cross check

In the boundary cross check scenario, the purpose is to ensure that memory accesses stay within the defined boundaries of a specific region. The check involves verifying whether the correct region is being accessed by the master device or component. However, it's important to consider two additional factors: the start address of the transaction and the data length.

Even if the master correctly identifies and targets the intended region, a boundary cross error can occur if either the start address of the transaction or the combination of the start address and data length causes the transaction to cross the boundary of that region.

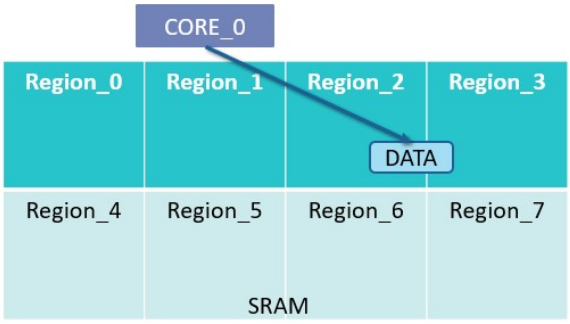

Figure 4.13: Boundary Cross check

Let's illustrate this with an example: Suppose there is a memory region with a start address of 0x1000 and an end address of 0x2000. If the master device attempts to access this region, but the start address of the transaction is 0x1FF0 and the data length is such that it extends beyond the region's boundary, a boundary cross error will be triggered.

The purpose of this check is to prevent unintended access beyond the boundaries of a region. Crossing these boundaries could lead to data corruption, security vulnerabilities, or unexpected system behavior. By reporting the boundary cross error, the MPU (Memory Protection Unit) or the system can take appropriate actions to

handle or prevent such scenarios. These actions may include halting the transaction, generating an interrupt, or notifying the master device about the error so that it can be addressed accordingly.

Snippet of UVM Code for Boundary cross:

```
class boundary_check_test extends mpu_test;
  typedef boundary_seq#(mpu_params) boundary_seq_t;
  typedef mpu_packet#(mpu_params) mpu_packet_t;

  boundary_seq_t boundary_seq_h;
  `uvm_component_utils(boundary_check_test)

  function new(string name = "boundary_check_test",uvm_component
   parent);
    super.new(name,parent);
  endfunction : new
  virtual function void build_phase(uvm_phase phase);
    boundary_seq_h = boundary_seq_t::type_id::create("
   boundary_seq_h");
    super.build_phase(phase);
endfunction : build_phase
virtual task run_phase(uvm_phase phase);
    boundary_seq_h.mpu_base_class = mpu_base_class;
  endtask : run_phase
endclass: boundary_check_test
```

Listing 4.5: Boundary Cross

Description:

In the given scenario, there is a waveform 4.14 representing the access of the second region of SRAM by CORE_0 in a system. The region is defined by a range of addresses from 32'h20000000 to 32'h2FFFFFFF, representing region number 2.

For a specific transaction initiated by CORE_0, the start address is 32'h2FFFFEFF, and the data length is 12'h2FF. To determine the end address,

we add the start address and the data length, resulting in 16'h300001FE.

Since the end address, 16'h300001FE, exceeds the upper boundary of the given region (32'h2FFFFFFF), the transaction crosses the boundary. This means that the transaction initiated by CORE_0 extends beyond the allowed memory range for region number 2.

When a transaction crosses the boundary of a memory region, a boundary cross signal is triggered and becomes high. This signal is detected by the MPU (Memory Protection Unit) in the system, which is responsible for monitoring and enforcing memory access rules.

Boundary Cross check waveform:

Figure 4.14: Boundary Cross check

With reference to waveform 4.14, upon detecting the high boundary cross signal, the MPU recognizes that the transaction has violated the memory region's boundaries. As a result, the MPU reports an error, indicating that the transaction initiated by CORE_0 has exceeded the permitted limits of the memory region.

The system or master device is alerted by the error that the MPU reports that the accessible memory has crossed the boundaries. The acceptable responses to this error may include stopping the transaction, stopping the operation, or alerting the relevant components to the violation, allowing for additional handling or problem-solving.

4.3 Formal Verification of MPU

SystemVerilog assertions are developed to validate each of the five checks as part of the MPU's formal verification procedure. By establishing an equivalent model for the MPU through these claims, its functionality may be verified.

The objective of writing assertions for formal verification is to account for all potential inputs and guarantee that the MPU's behaviour adheres to the expected requirements. You are effectively defining the intended attributes and limitations that the MPU should follow by asserting all tests.

Following the writing of the assertions, coverage code is often added to monitor the verification process. By counting the number of assertions that pass or fail, coverage code aids in assessing how thorough the verification process was. It offers a measure to evaluate how successfully the MPU checks have been verified.

4.3.1 Assertions for MPU Checks

```
1 asrt_check_vm_error_status : assert property (
2     @(posedge Clk)
3     disable iff (reset)
4     (vm[8:0] == errorID_status[8:0]) & config_reg |-> ##4
    Type_status_reg[0] );
```

Listing 4.6: Assertion for VM match/mismatch

```
1 asrt_check_mem_mismatch_status : assert property (
2     @(posedge Clk)
3     disable iff (reset)
4     invalid_trans & config_reg |-> ##4 mem_mismatch );
```

Listing 4.7: Assertion for Memory match/mismatch

```
asrt_check_write_acc_status : assert property (
    @(posedge Clk)
    disable iff (reset)
    (master_id[5:0] == errorID_status[14:9] & config_reg |-> ##4
   Type_status_reg[1] );
```

Listing 4.8: Assertion for write access check

```
 asrt_check_multi_region_access_status : assert property (
    @(posedge Clk)
    disable iff (reset)
    (vm[8:0] == errorID_status[8:0]) & config_reg |-> ##4
   Type_status_reg[1] );
```

Listing 4.9: Assertion for multi region access

```
asrt_check_boundary__status : assert property (
    @(posedge Clk)
    disable iff (reset)
    invalid_trans & config_reg |-> ##4 boundry_nok );
```

Listing 4.10: Assertion for boundary check

4.3.2 Assertions Results

Description:

As assertions 4.15 is successful, the MPU tests have been successfully checked for all potential inputs, satisfying the given requirements and limitations. Passing assertions imply that the MPU's behaviour fits the assertions' description of the MPU's behaviour.

It is crucial to remember that gaining code coverage and passing assertions do not ensure the exclusion of all potential errors or corner cases. Even though formal

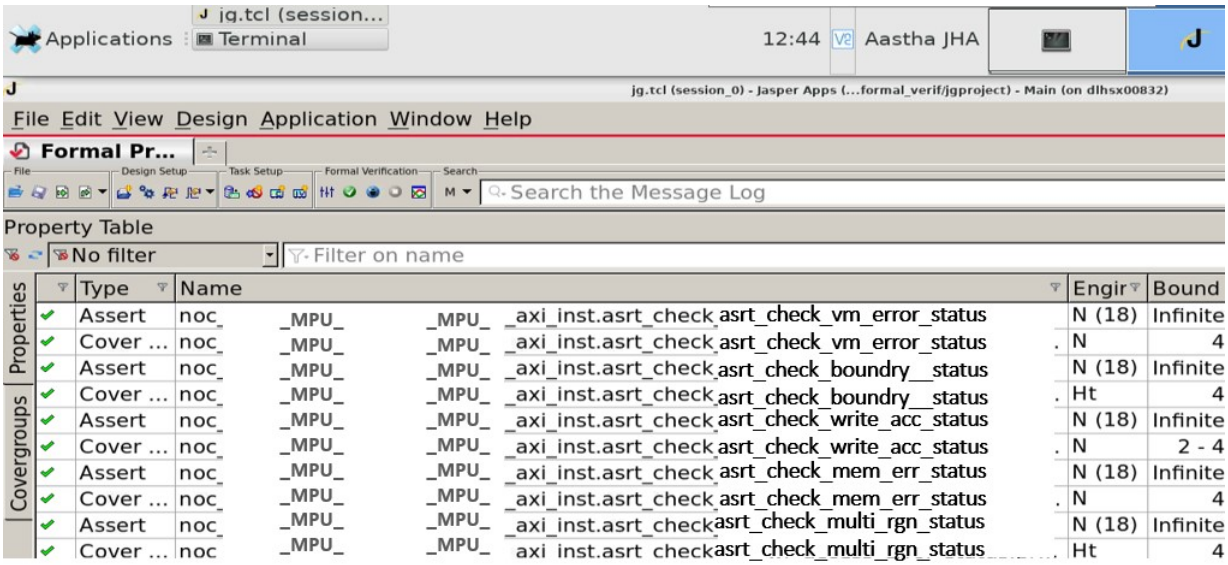

Figure 4.15: Assertions Results

verification is a potent tool, it is still important to take into account other types of testing, including functional testing, to guarantee the overall accuracy and reliability of the MPU implementation.

4.3.3 Coverage Results

After writing the assertions for the MPU and performing coverage analysis, it is found that the coverage result is 91 percent, as shown in the figure provided. This coverage result indicates the percentage of assertions that have been covered during the verification process. In other words, it represents the extent to which the behavior of the MPU has been exercised and verified.

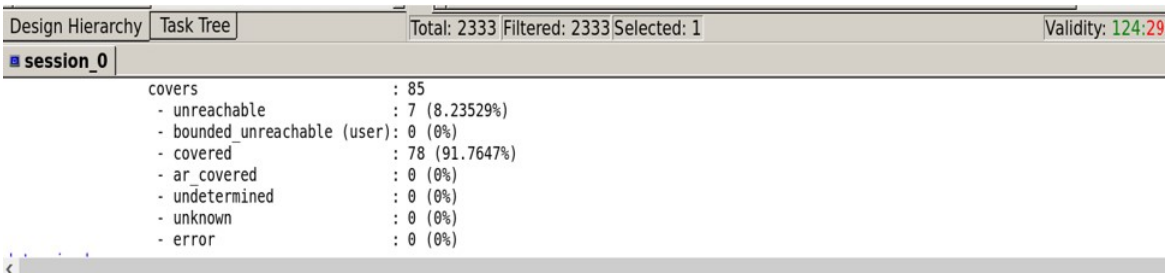

Figure 4.16: Coverage Results

According to Figure 4.16, since the coverage result is 91 percent, it suggests that there are still some assertions missing or not fully covered. To improve the coverage and ensure a more thorough verification, additional assertions need to be added.

To increase the coverage, modifications can be made to the existing assertions or new assertions can be introduced. These additional assertions should aim to cover the remaining unverified aspects of the MPU, including checks and other block behaviors that have not been adequately exercised [17].

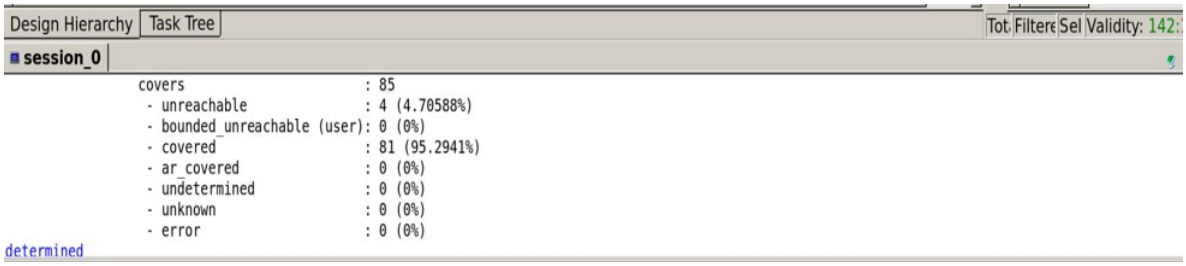

Figure 4.17: Modified Coverage Results

By modifying the existing assertions and adding new ones, the coverage result can be improved to 95 percent, as shown in the provided figure 4.17. This signifies that a larger portion of the MPU's behavior has been successfully verified.

However, the future aim in the MPU coverage analysis is to achieve even higher coverage levels of 97-98 percent. This means that further efforts are required to add more assertions and enhance the verification process. The additional assertions should focus on addressing the remaining unverified scenarios, edge cases, and potential corner cases that might impact the behavior of the MPU.

By continuously refining and expanding the set of assertions, as well as closely analyzing the coverage results, the aim of achieving 97-98 percent coverage can be pursued. This ongoing verification process helps in ensuring the correctness, robustness, and reliability of the MPU implementation.

4.4 Summary

For the Memory Protection Unit (MPU) in automotive microcontrollers to perform correctly and dependably, both functional and formal verification are essential. To ensure that the MPU's functionality, adherence to requirements, and effective application of memory protection rules are all present and valid, functional verification entails comprehensive testing of the MPU's behaviour in a variety of scenarios. The development and execution of test cases that cover various access patterns, transaction kinds, and error circumstances are part of the verification process.

On the other hand, formal verification makes use of mathematical modelling and analysis methods to formally demonstrate the accuracy and consistency of the MPU design. It entails creating claims, properties, and invariants that describe the desired behaviour of the MPU and the lack of mistakes or weaknesses. To thoroughly examine the design and find any violations or corner cases, formal verification techniques like model checking and symbolic execution are used.

A thorough method for validating the behaviour of the MPU and guaranteeing its dependability, safety, and security is provided by the combination of functional and formal verification. Potential problems including boundary crossings, unauthorised access, and data corruption can be recognised and dealt with by doing exhaustive testing, analysis of claims, and formal verification. The MPU is appropriate for use in automotive microcontrollers for applications like Advanced Driver Assistance Systems (ADAS) because of the verification process' contribution to the MPU's overall quality and dependability.

Chapter 5

Conclusion

5.1 Work Conclusion

In conclusion, a crucial step in ensuring the dependability, safety, and compliance of these systems is the verification of the AXI (Advanced eXtensible Interface) protocol in automotive microcontrollers. Microcontrollers are significantly used in the automobile sector to manage numerous electronic systems in cars. These microcontrollers frequently employ the AXI protocol to streamline communication between various parts, including CPUs, memory, and peripherals. To confirm that the AXI protocol complies with the unique specifications and norms of the automobile industry, thorough testing and validation are performed on automotive microcontrollers. This verification procedure is necessary to identify and address any potential problems that could result in functional errors, security flaws, or safety risks. Potential problems, such as data integrity concerns, timing infractions, or interoperability difficulties, can be found and resolved early in the development cycle by correctly validating the AXI protocol in automotive microcontrollers. This lowers the likelihood of malfunctions and ensures correct operation in actual vehicle conditions, improving the microcontroller's overall reliability and safety.

To make sure the MPU checks are accurate and consistent, formal verification approaches including assertion-based verification and mathematical proof are used. Formal verification aids in demonstrating that implemented checks correspond to the

stated requirements and are in compliance with applicable automotive standards, laws, and safety guidelines. Verifying the MPU checks in automotive microcontrollers can be done in a strong and all-encompassing way thanks to the combined efforts of functional and formal verification. Potential problems like unauthorised access, memory corruption, or security vulnerabilities can be found and fixed early in the development process by checking the MPU checks.

To confirm that the MPU checks work as intended, the verification method entails creating and writing assertions, running test cases, analysing coverage, and conducting thorough testing. These efforts are supplemented by formal verification approaches, which offer logical justification and mathematical proofs to ensure the accuracy and integrity of the implemented checks. The safe and secure operation of crucial systems within automobiles is aided by the verification of MPU checks in automotive microcontrollers. It helps the industry satisfy standards like ISO 26262 for functional safety. Automotive microcontrollers can work dependably in real-world circumstances by successfully confirming the MPU tests, providing the essential protection for crucial functions, reducing risks, and guaranteeing the general safety and security of vehicles and their occupants.

In conclusion, a critical step in ensuring the accurate and secure operation of these systems is the functional and formal verification of MPU checks in automotive microcontrollers. Potential problems are found and addressed by thorough testing, analysis, and formal verification procedures, guaranteeing that the microcontroller complies with safety regulations and meets the demanding standards of the automotive industry.

5.2 Future Scope of Work

Achieving higher coverage percentages of 97-98 percent is the long-term goal of the MPU coverage analysis. This necessitates further work to improve the verification process by introducing more assertions and addressing the remaining unresolved situations, edge cases, and hypothetical corner cases that might affect the MPU's

behaviour. The emphasis will be on continuously growing and improving the collection of assertions in order to cover a greater variety of test situations in order to meet this objective. The various memory access patterns, transaction kinds, and error scenarios are taken into account in this. The overall coverage level can be raised by paying close attention to the aspects that have low coverage and thoroughly analysing the coverage data to find those areas. The goal is to guarantee the accuracy, dependability, and robustness of the MPU implementation through these ongoing testing activities. In order to reduce potential hazards and vulnerabilities in automotive microcontrollers used in ADAS applications, better coverage levels must be attained. This will increase confidence in the MPU's performance.

References

[1] R. Watn, T. Njolstad, F. Berntsen, and J. F. Lonnum, "Independent clocks for peripheral modules in system-on-chip design," in *IEEE International [Systems-on-Chip] SOC Conference, 2003. Proceedings.* IEEE, 2003, pp. 25–28.

[2] R. Dorsch, R. H. Rivera, H.-J. Wunderlich, and M. Fischer, "Adapting an SoC to ATE concurrent test capabilities," in *Proceedings. International Test Conference.* IEEE, 2002, pp. 1169–1175.

[3] K. Pooja, S. Krishnakumar, and H. R. Aradhya, "Verification of interconnection IP for automobile applications using system verilog and UVM," in *2018 3rd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT).* IEEE, 2018, pp. 1119–1123.

[4] A. B. Mehta, "ASIC/SoC functional design verification," *Publ. Springer*, 2018.

[5] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L.-C. Wang, "Challenges and trends in modern SoC design verification," *IEEE Design & Test*, vol. 34, no. 5, pp. 7–22, 2017.

[6] B. Wang and Z. Lu, "Efficient support of AXI4 transaction ordering requirements in many-core architecture," *IEEE Access*, vol. 8, pp. 182 663–182 678, 2020.

[7] R. Pan, G. Peach, Y. Ren, and G. Parmer, "Predictable virtualization on memory protection unit-based microcontrollers," in *2018 IEEE Real-Time and Em-*

bedded Technology and Applications Symposium (RTAS). IEEE, 2018, pp. 62–74.

[8] R. Stajnrod, R. Ben Yehuda, and N. J. Zaidenberg, "Attacking trustzone on devices lacking memory protection," *Journal of Computer Virology and Hacking Techniques*, pp. 1–11, 2021.

[9] F. Paci, D. Brunelli, and L. Benini, "Lightweight IO virtualization on MPU enabled microcontrollers," *ACM SIGBED Review*, vol. 15, no. 1, pp. 50–56, 2018.

[10] D. Verhaert, "An architecture-agnostic memory protection interface for the tock operating system," 2018.

[11] A. Burkert, "Perspectives of software-based connectivity," *ATZelektronik worldwide*, vol. 8, no. 1, pp. 10–13, 2013.

[12] S. A. Saji and K. Sivasankaran, "Test suite for SoC interconnect verification," in *2017 International conference on Microelectronic Devices, Circuits and Systems (ICMDCS)*. IEEE, 2017, pp. 1–6.

[13] D. Saha and S. Sur-Kolay, "SoC: a real platform for IP reuse, IP infringement, and IP protection," *VLSI Design*, vol. 2011, pp. 1–10, 2011.

[14] 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," in *2012 International SoC Design Conference (ISOCC)*. IEEE, 2012, pp. 175–178.

[15] F. A. Da Silva, A. C. Bagbaba, S. Hamdioui, and C. Sauer, "Efficient methodology for ISO26262 functional safety verification," in *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, 2019, pp. 255–256.

[16] B. Wile, J. Goss, and W. Roesner, *Comprehensive functional verification: The complete industry cycle.* Morgan Kaufmann, 2005.

[17] W. Yang, M.-K. Chung, and C.-M. Kyung, "Current status and challenges of SoC verification for embedded systems market," in *IEEE International [Systems-on-Chip] SOC Conference, 2003. Proceedings.* IEEE, 2003, pp. 213–216.