Design and Verification of IDI VIP for Cache Coherency Management

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

MASTER OF TECHNOLOGY

in VLSI DESIGN

Submitted By

RISHABH KHANNA

602362027

Under Supervision of

Dr. Dinesh

Assistant Professor

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

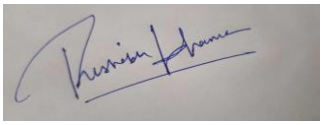
THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY

(A DEEMED TO BE UNIVERSITY), PATIALA, PUNJAB

JUNE, 2025

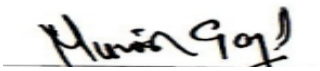
DECLARATION

I, **Rishabh Khanna** hereby declare that the work presented in this report entitled **"Design and verification of IDI VIP for Cache Coherence Management"** in partial fulfilment of the requirement for the award of degree of **Master of Technology (VLSI Design)** submitted at Electronics & Communication Department**,** Thapar Institute of Engineering & Technology (Deemed to be University), Patiala is an authentic record of work carried out under the supervision of **Dr. Dinesh (Assistant Professor, Electronics & Communication Department, Thapar Institute of Engineering & Technology)** and under Industrial mentor **Mr. Adrian Cretzu** (Tech Lead, Intel Corp) from June-2024 to July 2025. The matter presented in this has not been submitted in part or full to any other institute for the award of any other degree.



Date: 30 June 2025

Rishabh Khanna
602362027

 Mr. Munish Goyal Director of Engineering, Intel Corporation Date:30/06/2025	 Dr. Dinesh Assistant Professor Department of Electronics and Communication Engineering Thapar Institute of Engineering & Technology Patiala, Punjab Date:30/06/2025

CERTIFICATE

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

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

To Whomsoever It May Concern

WWID: 12268089

Employee Name: Rishabh Khanna

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

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

We wish you all the best for your future assignments.

Yours Sincerely

Gowre Saseedaran

Date: June 27, 2025

Place: **Bangalore**

ACKNOWLEDGEMENT

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

I would like to express deep gratitude to my industry mentor, **Mr. Adrian Cretzu**, Tech Lead, Intel Corporation, for all his guidance, unwavering support and feedback, through which I was able to push my limits and grow. I would like to further extend my thanks to **Mr. Munish Goyal**, Director of Engineering, Intel Corporation, for his valuable advice and guidance. Further, I extend my gratitude to **Dr. Kulbir Singh**, Head of Department, Electronics and Communication Engineering, Thapar Institute of Engineering & Technology for providing me this opportunity. At last, I would like to thank my family, friends and colleagues for their support and valuable inputs.

ABSTRACT

This report focuses on the design and verification of a Verification IP (VIP) for the Interconnect Direct Interface (IDI) protocol, specifically aimed at managing cache coherence in multi-core systems. Cache coherence is a critical aspect of modern computer architectures, ensuring that multiple processors maintain a consistent view of memory. The IDI protocol plays a vital role in facilitating efficient communication and data consistency across different cache levels.

The primary objective of this research is to develop a robust and comprehensive VIP that accurately models the IDI protocol's behaviour and verifies its cache coherence mechanisms. The methodology involves the creation of a Bus Functional Model (BFM) that simulates IDI protocol transactions and interactions. This BFM is then integrated into a Universal Verification Methodology (UVM) environment to perform extensive verification and validation.

Key findings from this research demonstrate that the developed VIP effectively identifies and resolves cache coherence issues, providing a reliable tool for hardware designers to validate their implementations. The VIP's ability to simulate various cache coherence scenarios and detect protocol violations significantly enhances the verification process's efficiency and accuracy.

In conclusion, this thesis presents a detailed account of the design and verification process for an IDI protocol VIP, emphasizing its importance in ensuring cache coherence in multi-core systems. The developed VIP serves as a valuable resource for future research and development in hardware verification, contributing to the overall reliability and performance of advanced computing systems.

TABLE OF CONTENTS

Sr. No	Name of the Chapters	Page No
	Pre-pages……	
	Declaration……	*ii*
	Acknowledgement……	*iii*
	Abstract……	*iv*
	Table of Content ……	*v*
	List of Figures……	
Chapter 1	Introduction to Verification……	*1-5*
1.1	Levels of Verification…….....	*1*
1.2	Types of Verification…….....	*2*
1.3	Verification Cycle …….....	*3*
1.4	Verification Challenges.....	*4*
Chapter 2	Literature Survvey.....	*5-6*
Chapter 3	Cache Coherency and Verification in SOC Design.....	*7-20*
3.1	Introduction.....	*7*
3.2	Cache Write Policies.....	*7*
3.3	Cache Mapping Techniques.....	*8*
3.4	MESI Protocol.....	*11*
	3.4.1 State Diagram of MESI Protocol.....	*10*
	3.4.2 Snooping.....	*12*
3.5	MOESI Protocol.....	*12*
3.6	Universal Verification Methodology.....	*13*
	3.6.1 UVM Factory and Configuration.....	*16*
	3.6.2 UVM Reporting and Messaging.....	*17*
	3.6.3 UVM Phasing.....	*17*
3.7	UVM and SystemC Integration.....	*18*
	3.7.1 Key Concepts in UVM- SystemC Integration.....	*19*
	3.7.2 Steps for UVM-SystemC Integration.....	*19*
	3.7.3 Challenges and Ongoing Research.....	*20*
Chapter 4	Results and Discussion.....	*22*
4.1	UVM-SystemC Environment Verification.....	*21*
4.2	Verification of Virtual Channel Classes.....	*24*
4.3	Verification of Uncore to Memory Interface.....	*24*
	4.3.1 Interface Development using XML and TDIF.....	*25*

Sr. No	Name of the Chapters	Page No
Chapter 4	Results and Discussion	
4.4	Memory Development for Uncore BFM.....	*28*
4.5	Uncore Verification Component Development.....	*30*
4.6	Development of Unique ID manager in BFM.....	*32*
4.7	Cache Memory Development for Core BFM.....	*33*
4.7	Discussions.....	*35*
Chapter 5	Conclusion & Future Scope ……	*37-38*
	References…….....	*39-41*

LIST OF FIGURES

Sr. No	Figure Details	Page No
Figure 1.1	*Design Cycle of an IC* ……	*1*
Figure 2.1	*SOC Division for Verification*……	*2*
Figure 3.1	*Coverage vs Time in Direct Stimuli Verification*……	*2*
Figure 1.4	*Coverage vs Time in Constraint Random Verification*……	*3*
Figure 3.1	*Cache Coherence Problem*……	*8*
Figure 3.2	*Cache Mapping Process*……	*9*
Figure 3.3	*Cache Organization*.....	*10*
Figure 3.4	*Direct Mapped Cache*.....	*11*
Figure 3.5	*Set- Associative Mapped Cache*.....	*11*
Figure 3.6	*State Diagram of MESI Protocol*.....	*15*
Figure 3.6	*UVM Testbench Architecture*……	*17*
Figure 3.4	*Phases in UVM*……	*20*
Figure 3.5	*UVM-SystemC Integration*……	*22*
Figure 4.1	*Tracker logs of cache states*……	*25*
Figure 4.2	*Performing Write Operation in UVM SC Environment*……	*26*
Figure 4.3	*Performing Read Operation in UVM-SC Environment*……	*22*
Figure 4.4	*Execution Log of VCHAN Test Cases for Core-to-Uncore Request Interface*	*26*
Figure 4.5	*Data Interface Model for Fabric to Memory Transactions*……	*27*
Figure 4.6	*Execution Log of Interface Test Cases for Fabric-to-Memory Request and Response*……	*28*
Figure 4.7	*Architecture of Interface between Memory and Uncore*.....	*29*
Figure 4.8	*End to End Simulation Test in UVM-SC*……	*30*
Figure 4.9	*Architecture of Verification Component*……	*32*
Figure4.10	*Structure of Core Cache Model in XML Format*.....	*34*

LIST OF ABBREVIATIONS

1. SoC: - System On Chip
2. IC - Integrated Circuit
3. IP - Intellectual Property
4. RTL - Register Transfer Level
5. SV - System Verilog
6. UVM - Universal Verification Methodology
7. VIP - Verification Intellectual Property
8. BG - Block Guide
9. SVA - System Verilog Assertions
10. PCIe - Peripheral Component Interconnect Express
11. DMA - Direct Memory Access
12. UART - Universal Asynchronous Receiver-Transmitter
13. GPIO - General-Purpose Input/Output
14. SISO - Single Input Single Output
15. TLM - Transaction-Level Modelling
16. HLS - High-Level Synthesis
17. BFM - Bus Functional Model
18. DUT - Design Under Test
19. UVC - Universal Verification Component
20. FPGA - Field-Programmable Gate Array
21. MESI - Modified, Exclusive, Shared, Invalid
22. F - Forward (in MESI-IF protocol)
23. I - Intermediate (in MESI-IF protocol)
24. AI - Artificial Intelligence
25. API - Application Programming Interface

CHAPTER 1
INTRODUCTION TO VERIFICATION

Verification is a procedure to check and confirm the functional correctness of a design against specified specifications, ensuring the design is functionally sound. The design of a System on Chip (SoC) is highly complex, making it challenging to produce a bug-free design. Consequently, defects are often detected during the early stages of semiconductor design verifications. Figure 1.1 illustrates the design cycle of an Integrated Circuit (IC)[1].

Figure 1:1 Design Cycle of an IC

1.1 LEVELS OF VERIFICATION

Verification is a significant challenge for companies working on IP and SoC products to meet customer requirements. The complexity of modern circuits leads to extended verification times, which can only be managed through advanced verification techniques and optimal reuse. Approximately 70% of the entire chip design time is devoted to verification. Before moving the expensive chip to manufacturing, all bugs and errors in the design must be eradicated. The levels of verification are:

A. Intellectual Property (IP)/Module Level:

The RTL of this IP offers extensive configurability through its generic parameters. To ensure smooth integration into larger systems such as SoCs or subsystems, the IP undergoes regression verification across all parameter-defined configurations. This verification is performed at the IP level within a dedicated System Verilog/UVM test environment [2],[3], aiming for complete coverage.

B. Subsystem/Platform Level:

The main goal is to make sure all-important subsystems work correctly before they're combined into the final SoC. For example, within our team, we refer to one such subsystem as a "platform." This platform includes six processor cores along with

several other bus masters. It also has a bus matrix and multiple slave components. Before this platform becomes part of the full SoC, we carefully verify how all these masters and slaves interact with each other at the subsystem level [4] to ensure everything functions as expected.

C. SoC Level:

All non-critical or peripheral IP are integrated into the one single code SoC RTL, and this RTL code is verified by SoC verification engineers [5].

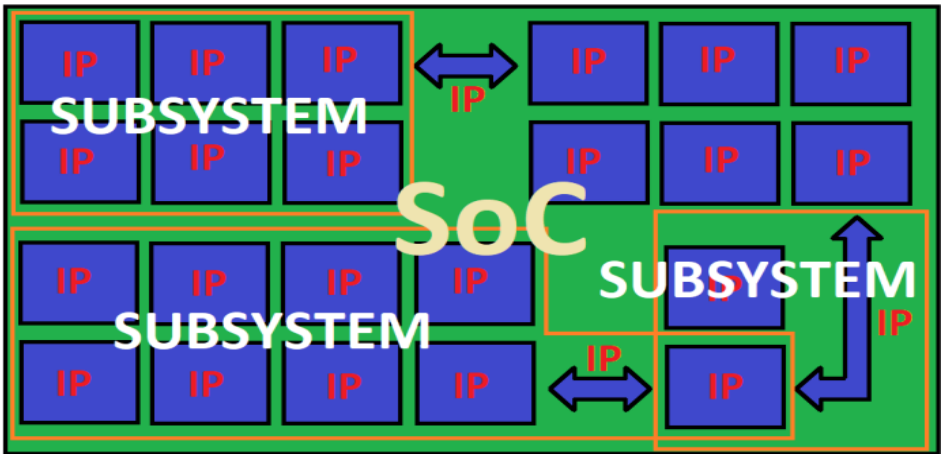


Figure 1.2: SOC Division for Verification

Figure 1.2 shows how chip verification is done in stages—starting at the IP level, moving to subsystems, and finally to the full SoC. Each stage ensures that smaller blocks work correctly before integrating them into the whole system.

1.2 TYPES OF VERIFICATION

1. **Direct Testing Verification:**

   Based on a specification list called Block Guide (BG), a verification plan is made, and targeted test cases are developed to check the IP's functionality. This ensures steady growth to 100% coverage. Fig 1.3 below shows Coverage vs Time plot for direct stimuli verification [6].

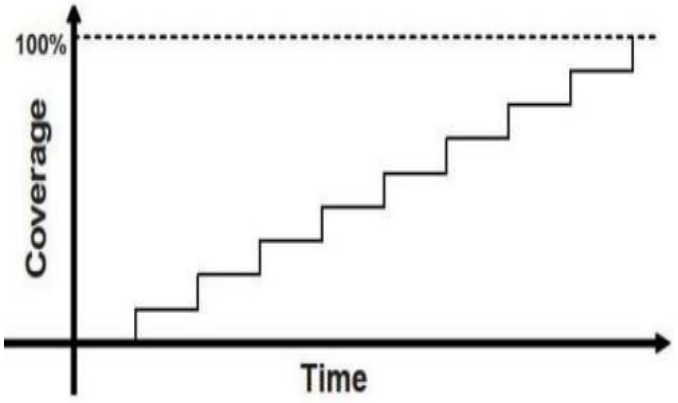


Figure 1.3: Coverage vs Time in Direct Stimuli Verification

2. **Constrained Random Stimulus Verification:**

Randomization is a method used in test cases where certain variables—marked as "rand"—are given different values each time the test runs. These values aren't chosen manually; instead, they're driven by a special seed value that the simulator creates randomly during execution. Because the seed changes with every run, the variables get new values each time, allowing the same test to produce different results. This helps uncover unexpected issues by exploring a wide range of scenarios automatically [7],[8].

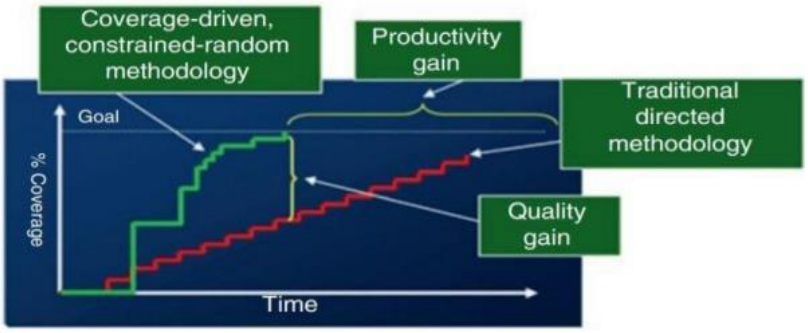


Figure 1.4: Coverage vs Time in Constraint Random Verification

3.Formal Verification:

In formal verification, inputs to the IP are generated randomly by tools, and outputs are checked against assertions written in System Verilog Assertions (SVA), a subset of System Verilog designed for this purpose. While formal verification provides rigorous checking, it has drawbacks.. Despite these challenges, formal verification remains a powerful method for ensuring design correctness, especially when complemented with other verification techniques to handle the intricacies of modern designs.\

4.VIP Based Verification:

VIPs are pre-verified components that simulate standard protocols and interfaces, such as PCIe, USB, Ethernet, etc. They provide a standardized way to verify the functionality of these interfaces within the SoC. VIPs can be integrated into the verification environment to simulate real-world scenarios, ensuring that the design meets the required specifications. This approach significantly reduces the time and effort needed for verification, as VIPs are reusable across multiple projects.

1.3 VERIFICATION CYCLE

Verification cycle generally consists of four phases:

1.Development

Involves verification plans, architecture, testbench, and sequences development. VIPs can be integrated at this stage to simulate standard protocols and interfaces.

2. **Simulation**

   Includes compilation, elaboration, and waveform generation. VIPs help in simulating real-world scenarios, providing a more comprehensive verification environment.

3. **Debugging**

   At transaction level, signal level, etc. VIPs can assist in identifying issues related to standard protocols and interfaces.

4. **Coverage**

   Functional, code, and SVA coverage, which feedback to the first stage. VIPs ensure that all standard protocols and interfaces are thoroughly tested [9], contributing to higher coverage.

Each phase plays a crucial role in reducing time and improving throughput and accuracy.

1.4 VERIFICATION CHALLENGES

Due to increasing complexity and market demands, engineers face additional pressure to complete complex projects quickly. Challenges include:

- **Production:**
  - Managing complex projects in a short period. Design engineers have benefit from advancements from transistor-level to system-level design methods engineers need similar benefits to handle increasing device complexity. VIPs provide a standardized and reusable approach, enhancing production efficiency.
- **Efficiency:**
  - Reducing human intervention in verification tasks. As machinery complexity increases, minimizing manual intervention is advisable. Paid authentication services and appropriate verification can reduce manual intervention. VIPs automate the verification of standard protocols, increasing efficiency.
- **Reusability:**
  - The ability to reuse the current verification environment or parts of it for new projects or later generations of the same project. This is achieved by developing a modular verification environment and better documentation. VIPs are inherently reusable, providing significant time and cost savings.
- **Completeness:**
  - Covering as much of the design functionality as possible. Improving productivity, efficiency, and reusability provides more time to focus on verification completeness. Introducing methodologies that focus on completeness and facilities that provide greater visibility to verification progress can further improve completeness. VIPs ensure that all standard protocols and interfaces are thoroughly tested.

CHAPTER 2
LITERATUR SURVEY

The verification of System on Chip (SoC) designs has become increasingly complex due to the integration of numerous components and the need for high reliability and performance. To address these challenges, advanced verification methodologies such as Universal Verification Methodology (UVM) and SystemC have been widely adopted. Verification Intellectual Property (VIP) plays a crucial role in this process, providing reusable and standardized components that facilitate efficient and thorough verification.

This literature survey aims to explore the current state of research in the field of UVM and SystemC-based verification, with a particular focus on the verification of VIP. The survey will cover various aspects, including the integration of UVM and SystemC, scalability issues, reusability of verification components, debugging techniques, and the application of advanced technologies such as AI and machine learning in verification.

S. Wu, K. Zhao, X. Wang, S. He, and D. Guo [4], in 2023, presented a UVM-based verification platform for hardware and software co-design. They discussed the challenges of integrating hardware and software verification and proposed a platform that enhances the interoperability between UVM and SystemC environments. Their approach led to improved verification efficiency and coverage, addressing the complexities of co-design verification.

S. A. Islam and S. Katkoori [7], in 2013, focused on accelerating System Verilog UVM-based VIP to improve the verification methodology for image signal processing designs using hardware emulators. They highlighted the benefits of using hardware emulation to speed up the verification process and demonstrated how this approach can enhance the overall quality and efficiency of the verification.

X. Su, H. Caceres [8], in 2022, developed a high-level verification flow for digital logic design based on high-level synthesis. The study emphasized the importance of integrating high-level verification techniques with traditional UVM methodologies to handle the increasing complexity of modern digital designs. The proposed flow improved the scalability and reusability of verification components.

G. Sharma, L. Bhargava [5], in 2024, explored the integration of AI and machine learning in UVM for enhanced low-power semiconductor design verification and testing. The research demonstrated how AI and machine learning techniques can be used to optimize the verification process, reduce power consumption, and improve the accuracy of verification results.

E. Massoud, M. AbdelSalam, M. Safar, and M. W. El-Kharashi [1], in 2022, proposed a reusable UVM-SystemC verification environment for simulation, hardware emulation, and FPGA prototyping. They presented case studies that showcased the versatility and effectiveness of their environment in various verification scenarios.

B. Sniderman and V. M. Yankelevich [12], in 2014, addressed the challenges of multi-language verification in real-world problems. They proposed solutions for integrating different verification languages, such as UVM and SystemC, to create a cohesive verification environment. Their work emphasized the importance of multi-language support in handling complex verification tasks and improving overall verification efficiency.

S. A. Islam and S. Katkoori [7], in 2018, focused on high-level synthesis of key-based obfuscated RTL datapaths at the 19th International Symposium on Quality Electronic Design (ISQED) in Santa Clara, CA, USA. They proposed a method for obfuscating RTL datapaths to enhance security against reverse engineering, showing how their approach can protect intellectual property in hardware designs

J. S. Barros, V. H. Schulz, and D. V. Lettnin [9], in 2018, proposed an adaptive closed-loop verification approach in UVM-SystemC for AMS circuits at the 31st Symposium on Integrated Circuits and Systems Design (SBCCI) in Bento Gonçalves, Brazil. They introduced a methodology that adapts to verification results in real-time, improving the verification process for analog-mixed-signal (AMS) circuits.

M. Mefenza, F. Yonga, and C. Bobda [10], in 2014, presented an automatic UVM environment generation for assertion-based and functional verification of SystemC designs at the 15th International Microprocessor Test and Verification Workshop in Austin, TX, USA. Their work automated the creation of UVM environments, enhancing the efficiency and effectiveness of SystemC design verification

X. Pan and B. Jonsson [11], in 2014, modeled cache coherence misses on multicores at the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) in Monterey, CA, USA. They developed predictive models to analyze cache coherence misses, providing insights into improving multicore processor performance

I Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt [12], in 2013, explored cache coherence for GPU architectures at the 19th International Symposium on High Performance Computer Architecture (HPCA) in Shenzhen, China. They proposed a cache coherence protocol tailored for GPUs, demonstrating how it enhances performance and efficiency in GPU computing

S. Kaxiras and A. Ros [13], in 2013, offered a new perspective for efficient virtual-cache coherence in the Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13). They introduced innovative techniques for managing virtual-cache coherence, showing improvements in system performance and scalability.

Nagarajan et al. [17] (2020) provides a comprehensive overview of memory consistency and cache coherence, essential concepts in computer architecture. Their work serves as a foundational text for understanding how memory operations are coordinated in multi-core systems.

Mengyao et al. [19] (2023) discuss a UVM-based reusable hardware accelerator verification platform at the International Conference on Integrated Circuits and Microsystems.

Their platform streamlines the verification of hardware accelerators, making it easier to reuse verification components across different projects.

Shukur et al. [20] (2020) explore coherence protocols in distributed systems, as published in the Journal of Applied Science and Technology Trends. Their study highlights the challenges and solutions associated with maintaining data consistency across distributed computing environments.

Tiwari [21] (2014) compares the performance of cache coherence protocols on multi-core architectures in a dissertation from NIT Rourkela. This work provides insights into how different protocols impact system performance, guiding the selection of appropriate coherence strategies for specific applications.

CHAPTER 3
CACHE COHERENCY AND VERIFICATION IN SOC DESIGN

3.1 INTRODUCTION

One of the biggest challenges in the contemporary design of System on Chip (SoC) is ensuring a consistent view of memory for multiple processors or cores. This problem is known as cache coherency: each core typically has its own cache that stores copies of the data from the main memory to reduce latency and improve performance. When one core changes a memory location, it must inform the other cores so that their views of memory are consistent. Failure in cache coherency makes different cores work on data that is either stale or unconsumable, which results in incorrect execution of programs and failures in the system.

Cache Coherency Problem

The cache coherency problem [10] is a fundamental issue in multi-core systems where each core has its own cache. Key issues in cache coherency include:

- **Data Inconsistency**: When multiple caches possess the same copies of a memory location, then updating one copy needs to reflect all other copies to make it non-inconsistent.
- **Write Propagation**: The write operation is being propagated from core to core in different caches.
- **Read/Write Synchronization**: It is about synchronizing read and write operations from multiple caches which ultimately lead to a common consistent memory state.

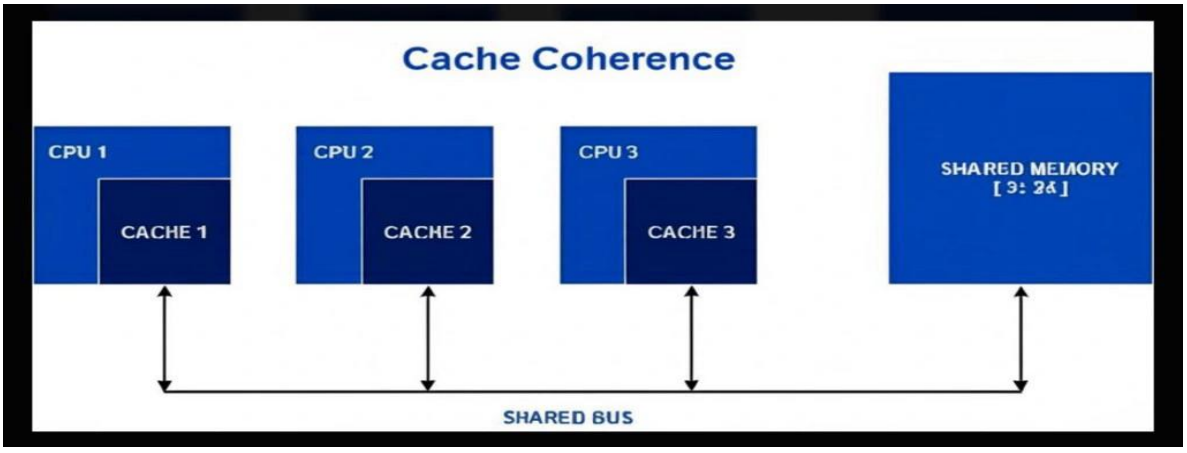


Figure 3.1: Cache Coherence Problem

As you can see in Figure 3.1, each CPU has its own cache, but they all share the same memory. This setup can lead to problems when one cache updates data and the others don't know about it—causing mismatches. That's what the cache coherence problem is all about—keeping everything in sync so all CPUs see the same data.

3.2 CACHE WRITE POLICIES

Caches are used because it stores the data temporarily and data can be accessed faster when compared to the main memory. But caches have limited size, so we need to have a proper

management polices to access the data so that we can write, read the data. There are 2 cache write policies:

- **Write through:** Here all the write operation are updated simultaneously to the cache and the main memory. This process is used when the write operations are less. This policy is more simpler but it has more delay as we have to update/write the data to both the locations i.e. cache and main memory. It helps in finding the recovery of data in case of system failure.
- **Write back:** Here in this method the data is updated in the cache every time when change occurs and memory at later time. The data is written or updated only in the cache and the cache block which is modified is updated to the main memory only when it's replaced.

Write back is more preferred because it has better performance. The main reason to use the cache is to avoid multiple times accessing to the main memory but whereas in write through policy, it is updated every time to the main memory. In write back policy, the data is only updated or written to the main memory when the cache line or cache block is ready to be replaced or invalidated. Write back saves a lot of bandwidth when compared to the write through caches[11]

3.3 CACHE MAPPING

Cache memory is a specialized form of high-speed storage that plays a crucial role in modern computing systems. It is designed to temporarily hold frequently accessed data and instructions, thereby speeding up data retrieval and enhancing overall system performance. By reducing the time needed to access data from the main memory, cache memory significantly improves the efficiency of processors, allowing them to execute tasks more rapidly.

The primary purpose of cache memory is to bridge the speed gap between the fast processor and the slower main memory. When a processor needs data, it first checks the cache memory. If the required data is found in the cache (a scenario known as a cache hit), it can be accessed much faster than if it had to be retrieved from the main memory. This reduces latency and increases the throughput of the system. Conversely, if the data is not in the cache (a cache miss), it must be fetched from the main memory, which takes more time.

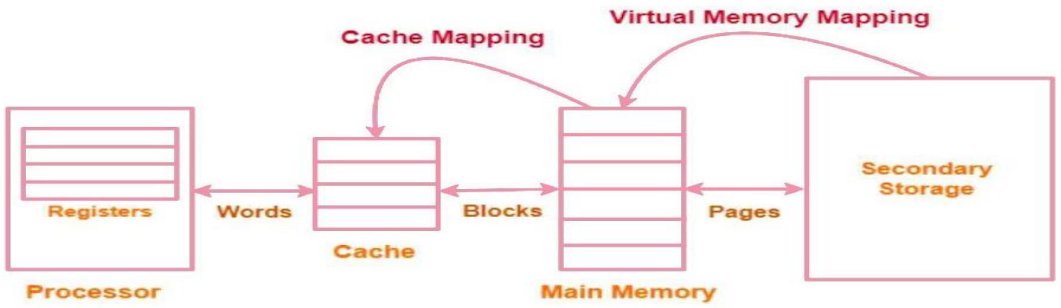


Figure 3.2: Cache Mapping Process

Figure 3.2 explains how data flows from the processor to cache and then to main memory or storage, helping speed up access through a process called cache mapping.

Types of Cache Memory:

Level 1(L1) Cache:

L1 cache is the smallest and fastest type of cache memory, located directly on the processor chip. It is divided into separate caches for instructions and data, known as instruction cache and data cache. Due to its proximity to the processor, L1 cache offers the quickest access times but has limited storage capacity. Its primary function is to provide immediate access to the most frequently used data and instructions, minimizing the time the processor spends waiting for data.

Level 2(L2) Cache:

L2 cache is larger than L1 cache and can be located on the processor chip or on a separate chip close to the processor. It serves as an intermediary between the L1 cache and the main memory, storing data that is not as frequently accessed as the data in L1 cache but still needs to be retrieved quickly. L2 cache helps reduce the frequency of accesses to the slower main memory, offering a balance between speed and storage size.

Level 3(L3) Cache:

L3 cache is the largest and slowest of the three cache levels, typically shared among multiple processor cores. It acts as a reservoir for data that is less frequently accessed but still benefits from faster access than main memory. L3 cache supports the L1 and L2 caches by storing additional data that might be needed by the processor. While L3 cache has the largest storage capacity, it is slower than L1 and L2 caches, providing a final layer of caching before data is accessed from the main memory.

This Figure 3.3 shows the three cache levels—L1 (fastest), L2, and L3 (largest)—which help the CPU quickly access data before checking slower main memory.

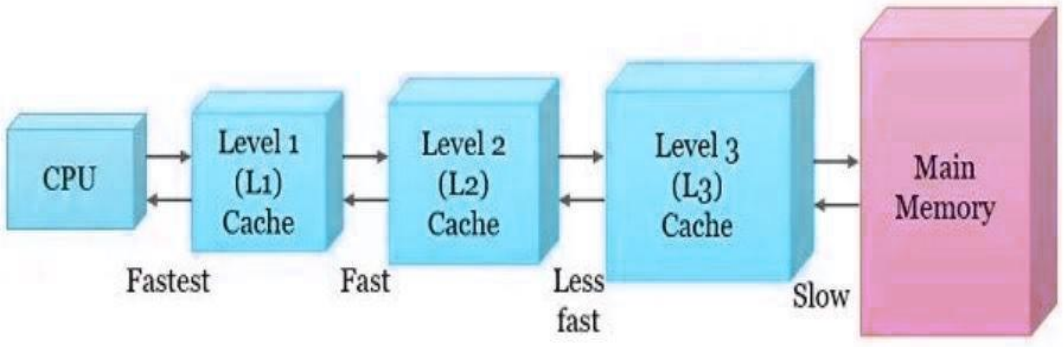


Figure 3.3: Cache Organization in Computer

There are different types of cache mapping techniques to load the data from the main memory which are as follows:

- 1. Direct mapping technique
- 2. Set associative mapping technique
- 3. Fully associative mapping technique

Direct Mapping: It maps all block of the main memory only to a single line of the cache. Say if the line has a data which is taken up by the main memory and if the new block arrives, the data

which is currently stored in the block will be replaced by the new data. This is the simplest technique as it requires less hardware. It has 2 fields: Tag and the Index field. The tag field is saved in the cache and the index is saved in the main memory. This direct mapping technique behavior is proportionate to the hit ratio. As each block of the main memory maps only to a single line of the cache, so if the new block arrives, the old block is trashed and if both the blocks are continuously eferenced, the 2 block will be swapped in and out continuously and it leads to the conflict miss.

Figure 3.4 shows how direct mapping works—each block from main memory maps to one fixed cache line. If a new block comes in, it replaces the old one, which can cause frequent swaps and conflicts if the same lines are accessed often.

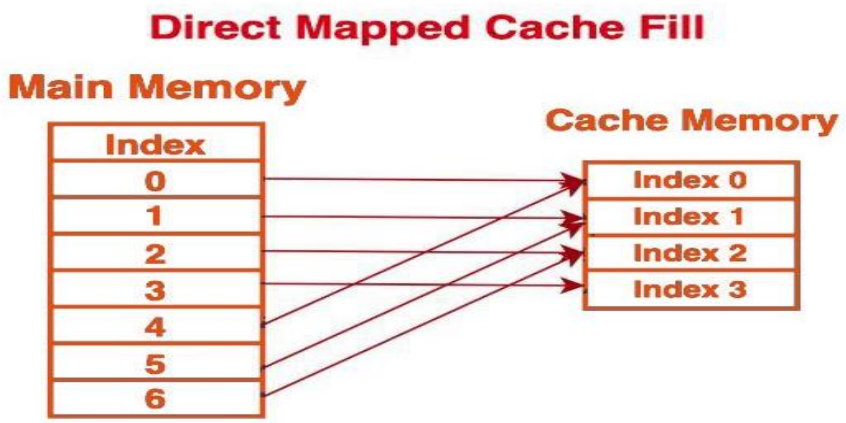


Figure 3.4: Direct Mapped Cache

Set-associative Mapping: This technique is introduced which decreases the drawbacks of the previous technique that is direct mapping. Here instead of a single line, we have set of lines which maps to the main memory block. Here the main memory block can be mapped to one of the lines of a particular set. The tag bit is to identify which of the particular set is used and the main memory block data is stored in the index address. Figure 3.5 shows how set-associative mapping works by grouping cache lines into sets. Unlike direct mapping, it gives more flexibility in placing data, reducing conflicts and improving cache performance.

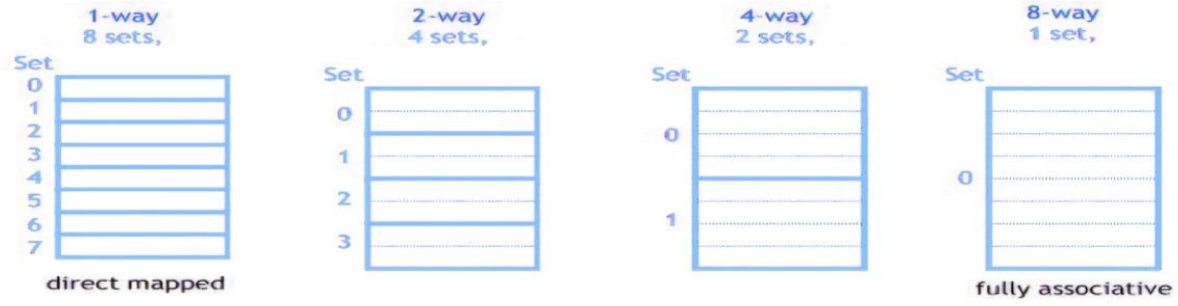


Figure 3.5: Set-Associative Mapped Cache

Fully associative mapping: In this technique, any block can be mapped to any line of the cache. It means that a single cache set with multiple cache lines. Here the index bit is used to identify

which data in the block is required and the tag is the rest bits. It is much flexible and has better hit rate than the other mapping techniques. But it takes more time as we need to search all the cache lines for a particular block in the cache, as it can be anywhere in the cache.

Necessity of Replacement Algorithms in Cache Memory Management:

As the foundation for effective data handling in computer systems, replacement algorithms are essential in the field of cache memory management. When the cache fills up, they are in charge of deciding which data should be deleted to make room for fresh data. The significance of these algorithms is highlighted by several important factors:

Cache Capacity Limitations

Because of financial and spatial limitations, cache memory is by nature smaller than main memory. As such, it is unable to hold all of the information that a processor may need. To keep the most relevant and often accessed data in the cache, replacement algorithms are crucial for handling this constrained amount of space.

Enhancement of Performance

Cache memory's main purpose is to speed up data access by keeping frequently used information nearer to the processor. By increasing cache hits—instances where the required data is located within the cache—replacement algorithms are essential for improving cache performance. These algorithms assist in maintaining high hit rates by carefully choosing which data to evict, which lessens the need for slower main memory and improves system performance in general.

Reduction of Latency

Replacement algorithms reduce the amount of time the processor needs to access data by keeping the most pertinent information in the cache. For applications like real-time processing, gaming, and high-performance computing that require quick data retrieval, this latency reduction is essential.

Improved Use of Resources

Effective use of the limited cache spac**e** is ensured by effective cache management using replacement algorithms. These algorithms improve resource utilization and system performance by removing less important data to make room for new data that is more likely to be accessed.

There are 3 main Cache Coherence Protocols:

3.4 MESI PROTOCOL

MESI protocol is the more widely used cache coherence protocol. It is an invalidate based cache coherence protocol and supports write back. This MESI protocol [12] decreases the number of times the main memory access compared to the MSI protocol. It has a dirty state present which shows that the data present in the cache is unlike from the main memory. It has 4 states:

- **Modified (M):** It means the cache line has been changed and is inconsistent with the main memory.
- It is the only cache who has the actual copy, and the data must be written back to the main memory before any changes take place.

- **Exclusive (E):** The cache line is the same as the main memory, and this cache has the only copy.
- **Shared (S):** The cache line is the same as the main memory, and other caches may hold copies of this data.
- **Invalid (I):** The cache line is invalid and does not hold valid data.

The MESI protocol uses these states to manage cache coherency through a series of state transitions triggered by read and write operations.

The MESI-IF protocol is an extension of the MESI protocol that includes additional states to handle more complex cache coherency scenarios. The additional states are:

- **Forward (F)**: Indicates that the cache line is shared and this cache is responsible for forwarding the data to other caches.
- **Intermediate (I)**: A transient state used during state transitions to handle intermediate steps in the coherency process.

The MESI-IF protocol provides finer control over cache coherency management, allowing for more efficient handling of complex multi-core interactions.

3.4.1 State Diagram of MESI Protocol

For the model, let us assume that RH be the Read hit of the cache and the RH occurs when the cache block is either in modified, shared or exclusive state. When the read operation of the cache occurs, it's stays in the same state as shown below in 3.4. It stays in the same state as long as the block is recent and so it is valid to read the same. If the block is in the shared state, then all the shared blocks are constant and memory read will not modify the content of the block. so therefore, it's not necessary to modify the state of the shared blocks that are present in other cores [13]. When the block is not present in the cache, then it means the state as invalid. Let RM be the Read miss, and when the block is in the invalid state, it is same as the cache miss of the read operation. To fulfil a read, the invalid block must be fetched from main memory and placed in the shared or exclusive states. Before doing so, the processor sends a message to the snoop bus called SHR(Snoop hit on a read). The other caches snoop the message and, depending on the state of the cache block, perform one of the following activities.

- **Exclusive:** One of the local caches has an exclusive cache block (multiple cache blocks cannot be in exclusive state). When the processor (which has that block in exclusive state) hears SHR, it replies by indicating that it shares the block and changes the state of its block from exclusive to shared. The starting processor will then perform a block read operation, which will alter the block's state from invalid to shared.
- **Shared:** The block is in the shared state in one or more caches. Hearing SHR, all of those processors send a signal to the originating processor which indicates that they are sharing the block. The initiating processor reads the block and changes the state from invalid to shared because the main memory copy is valid.

- **Modified:** In one of the local caches, one cache block has been modified (multiple cache blocks cannot be in modified state). The responding CPU sends a signal to the initiating processor when it hears the SHR, instructing it to try the read operation again later. The initiating processor stays in an invalid state and tries to read again later. Meanwhile, the responding processor changes the block's block status from modified to shared and updates the main memory copy of the block.

Because no other processor has a copy of this block, the SHR receives no signal. So the processor reads the block and updates the state to exclusive state.

Let us consider WH be the write hit of the cache and it occurs when the block is present in the cache. Write hit occurs either in Modified, Exclusive or Shared state. The block enters to the invalid state when write miss (WH) occurs.

- **Exclusive:** The block is only available to the beginning processor. The initiating processor just writes the block and switches the state from exclusive to modified.
- **Modified:** The block is exclusive to the initiating processor, but it is modified. The initiating processor simply writes the block and saves the state as the modified state.
- **Shared:** The block exists in one or more shared copies. A signal called invalidate transaction is sent by the processor. The processor with the shared block understand that the processor's plan to change the data when it receives this signal and those processors invalidate their blocks. And then the block is modified by the initiating processor, and the status is changed to modified.

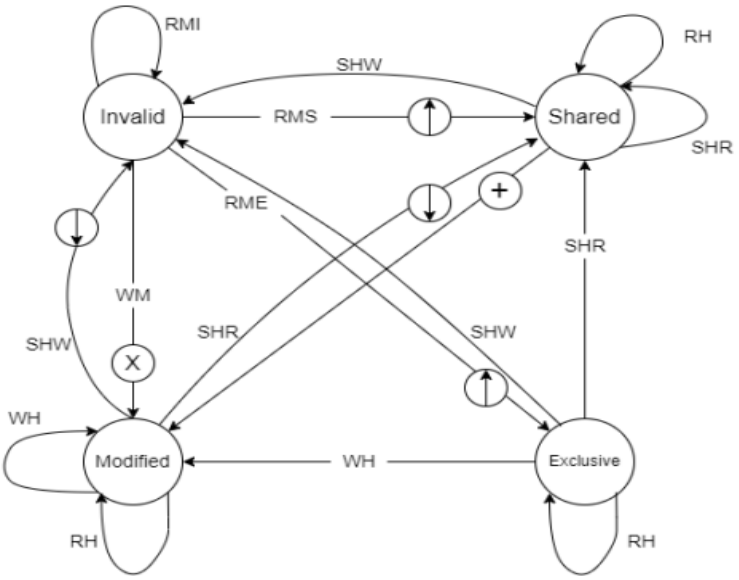


Figure 3.6: State diagram of MESI protocol

Figure 3.6 shows how a cache block moves between four states—Modified, Exclusive, Shared, and Invalid—based on read and write operations. It helps processors maintain data consistency when working with shared memory.

If the block is not available in the local cache, it is called write miss and is denoted as WM. When the block isn't in the local cache, it must be read from main memory and modified from there.

3.4.2 Snooping

Snooping is a technique used in cache coherency protocols to monitor and manage the state of cache lines. In a snooping-based system, each cache monitors (or "snoops") the bus for transactions that might affect the state of its cache lines. When a cache detects a relevant transaction, it updates its state accordingly. Snooping ensures that all caches are aware of changes to shared data, enabling them to maintain a consistent view of the memory.

3.4.3 MOESI PROTOCOL

The MOESI protocol is an extension of the MESI protocol, adding an additional state to improve efficiency in cache coherency. It stands for Modified, Owned, Exclusive, Shared, and Invalid. This protocol is used to manage the consistency of data stored in multiple caches in a multiprocessor system.

States and Transitions

The MOESI protocol includes five states:

1. **Modified (M)**: The cache line is modified and exclusive to the current cache. It has the most recent data.
2. **Owned (O)**: The cache line is modified and can be shared with other caches. The current cache is responsible for providing the data to other caches.
3. **Exclusive (E)**: The cache line is exclusive to the current cache but unmodified.
4. **Shared (S)**: The cache line is shared among multiple caches and is unmodified.
5. **Invalid (I)**: The cache line is invalid and does not hold valid data.

Benefits

The MOESI protocol offers several benefits over simpler protocols like MESI and MSI. One of the primary advantages is the efficient data sharing enabled by the Owned state. This state allows modified data to be shared directly between caches without writing back to main memory, reducing memory latency and improving performance.

Another benefit is the reduction in memory traffic. By minimizing the number of write-backs to main memory, the MOESI protocol reduces the load on the memory subsystem, leading to better overall system efficiency. This is particularly important in systems with high levels of data sharing and modification.

Challenges

Despite its advantages, the MOESI protocol also presents several challenges. One of the main challenges is the increased complexity of the protocol. The addition of the Owned state requires more sophisticated hardware and control logic, which can increase the cost and power consumption of the system. Another challenge is the coherence traffic generated by the protocol. Maintaining coherence

requires additional communication between caches, which can lead to performance bottlenecks, especially in systems with a large number of cores.

False sharing is another issue that can impact performance[14]. This occurs when different processors modify different variables that reside on the same cache line, leading to unnecessary invalidations and performance degradation.

3.6 UNIVERSAL VERIFICATION METHODOLOGY (UVM)

The Universal Verification Methodology, or UVM, is a structured approach created to make the verification process in System Verilog more efficient and organized. It comes with a set of predefined tools (APIs) and best practices that help engineers build strong and reusable test environments.

Interface: One important part of UVM is the use of interfaces. These are used to manage the communication between different levels of the testbench, from the top-level system to the individual components. Interfaces help keep things clean and reusable by grouping signals and making connections simpler. Just like modules, interfaces can be customized with parameters and can even include other interfaces inside them, making them very flexible for different design needs.

It's an open- source standard maintained by Accellera and can be freely acquired on their website. Figure 3.7 shows the structure of a UVM testbench.

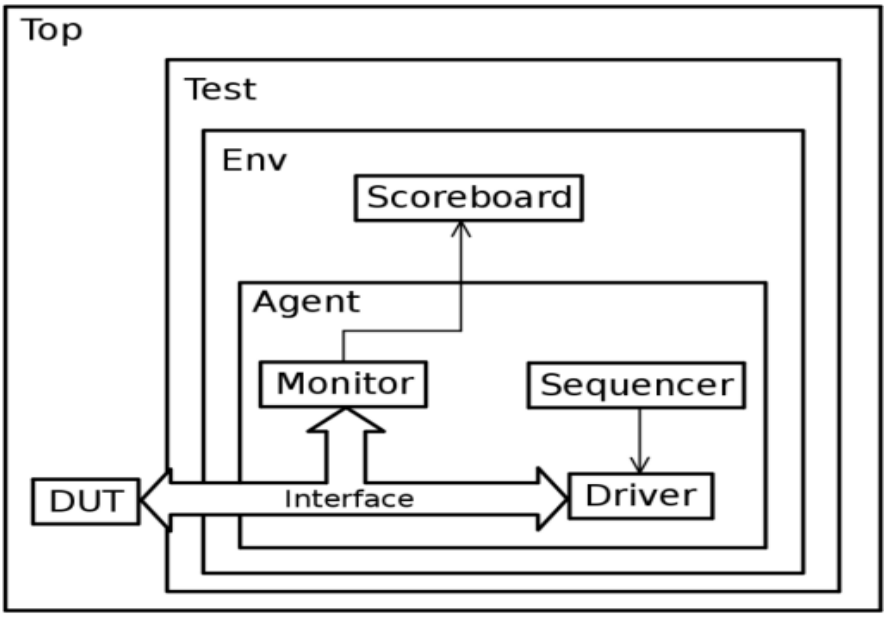


Figure 3.7: UVM Testbench Structure

UVM Sequence Item: A sequence item is like a data packet that gets sent to the DUT (Design Under Test). These are custom-made by extending the base UVM sequence item class, which itself is built on top of a UVM transaction. We create these user-defined sequence items mainly to take advantage of useful built-in functions like print, copy, and clone. These data fields are usually marked as random, with specific constraints added so the test input remains meaningful. By

randomizing the values within set limits, we can explore a wide variety of test scenarios with less manual effort.

UVM Sequence: A sequence is responsible for creating and sending a series of sequence items to the driver through the sequencer. To do this, we extend the UVM sequence class and specify what kind of sequence items it will handle. The sequence helps define how the test stimulus will behave, in what order data will be sent, and how interactions will happen with the DUT through the testbench. This helps simulate real-world conditions and traffic.

UVM Sequencer: The sequencer acts as the middleman between the sequence and the driver. It manages how the data flows between them using TLM (Transaction-Level Modeling) connections. The driver has a port, and the sequencer has an export—these are connected using the connect() method. We create a sequencer by extending the UVM sequencer class and passing the type of sequence item it will handle. The sequencer is a smart component—it can decide what kind of data items to send based on the DUT's condition by adjusting randomization settings. This helps make the test more adaptive and efficient.

UVM Driver: The driver is where test data becomes action. It takes the abstract sequence items and converts them into real signals or Bus Functional Models (BFMs) that can actually interact with the DUT. Drivers can work in two ways:

- Pull mode (default): where they request data from the sequencer,
- Push mode: where data is sent to them.

We build custom drivers by extending the uvm_driver class, which is a type of uvm_component. The driver is a key part of actually driving the test data into the DUT's interface.

UVM Monitor: The monitor plays the observer's role. It watches all the signal-level activity between the DUT and the testbench but never interferes. It captures data from the DUT's interface, converts it into standard sequence item formats, and forwards it to other components—like the scoreboard—for checking. Monitors are important for making sure the DUT is following the protocol and behaving correctly. Each agent usually includes its own monitor, and we create one by extending the uvm_monitor class.

UVM Agent: The agent acts like a team leader, bundling together the sequencer, driver, and monitor into a single unit. It's built by extending the uvm_component class. Agents help organize and manage verification more cleanly—especially when testing multiple protocols in a SoC-level environment.

Agents can be:

- Active: where they send and receive data to/from the DUT,
- Passive: where they only observe and report, without affecting the DUT.

Each agent has a build phase where components are created and a connect phase where everything is linked, but no run phase of its own.

UVM Scoreboard: The scoreboard is the checking mechanism. It keeps track of the input and output data of the DUT and compares them to make sure everything's working as expected. For

example, if a packet enters a device, the scoreboard ensures the correct packet also comes out. It uses inputs from monitors and checks them against what the output should be. If there's a mismatch, the scoreboard flags an error. It's one of the most critical components for verifying functionality and ensuring test accuracy.

UVM Environment: The UVM environment is the topmost layer in a testbench setup where different agents come together and work as a unit. It's created by extending the uvm_env class, which is itself based on the main UVM component class.

Along with agents, the environment can also include scoreboards that help check whether data is correctly transmitted from one end to the other, or compare DUT outputs against expected results from a reference model.

In more complex setups, this environment might also include other smaller environments that were tested earlier on their own—like at the block level—and are now being reused and integrated at the subsystem or full SoC level. This helps in scaling up verification in a modular and organized way.

UVM Test: This is the block that generates the testbench's test scenarios. Another task carried out by the test is to connect the sequencer with the sequences so that we can easily define which

sequence the sequencer will generate when a testcase is run. This allows the user to select the 30 kind of data items to transmit to the DUT based on their requirements without disrupting the sequencer or env code. User- defined tests are derived from the UVM test class, which is itself derived from the UVM component class. When the run test approach is used, the global run test assignment must be specified in the following underlying block.

UVM Top: It is the top block where DUT and test bench instances are built. This is the component that connects the DUT to the test bench. They will communicate using the virtual interface as a handshaking method. The interface is a module that contains all the DUT's signals. It contains all the SV files for various components, UVM packages, and sometimes certain assertion definitions, depending on the user's needs [15].

3.6.1 UVM Factory and Configuration

The UVM factory is a mechanism for creating and configuring objects dynamically. It allows for the substitution of components without modifying the testbench code, enhancing flexibility and reusability.

- **Factory Registration**: Components and sequences must be registered with the factory using the uvm_object_utils or uvm_component_utils macros. This registration enables the factory to create instances of the components and sequences.
- **Factory Overrides**: The factory allows for the substitution of components and sequences using type or instance overrides. Type overrides replace all instances of a component or sequence with a different type, while instance overrides replace specific instances.components. It allows for the centralized management of configuration settings, simplifying the process of configuring complex verification environments.

- **Setting Configuration**: Configuration settings can be set using the uvm_config_db:: set method. These settings can be applied to specific instances or to all instances of a component type.
- **Getting Configuration**: Configuration settings can be retrieved using the uvm_config_db: get method. Components typically retrieve their configuration settings during the build phase.

3.6.2 UVM Reporting and Messaging

UVM includes a robust reporting and messaging system to facilitate debugging and analysis. The system supports different message types (e.g., info, warning, error) and allows for filtering and customization of messages.

- **Message Types**: UVM supports several message types, including UVM_INFO, UVM_WARNING, UVM_ERROR, and UVM_FATAL. Each message type has a corresponding severity level.
- **Message Filtering**: Messages can be filtered based on their severity level and verbosity. This filtering allows users to control the amount of information displayed during simulation.
- **Custom Messages**: Users can define custom message types and formats using the uvm_report_message class. This customization allows for the creation of application-specific messages.

3.6.3 UVM Phasing

UVM introduces a phasing mechanism to control the execution order of various tasks in the verification environment. Phases include build, connect, run, and cleanup, ensuring that components are initialized, connected, and executed in a coordinated manner.

- **Build Phase**: Components are instantiated and configured during the build phase. This phase includes sub-phases such as build, connect, and end_of_elaboration.
- **Run Phase**: The run phase is the main simulation phase where stimulus is generated and applied to the DUT. It includes sub-phases such as reset, configure, main, and shutdown.
- **Cleanup Phase**: The cleanup phase is used to perform any necessary cleanup tasks after the simulation has completed. This phase includes sub-phases such as extract, check, and report.

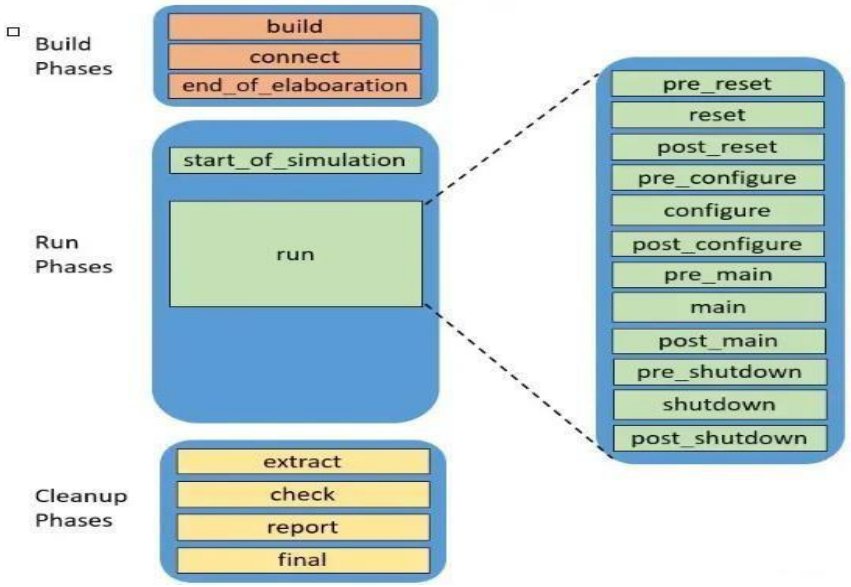


Figure 3.8: Phases in UVM

This diagram shows the different phases in UVM: build, run, and cleanup. Each phase has its own sub-steps, helping ensure everything in the verification process is set up, executed, and wrapped up in a proper order.

3.7 UVM AND SYSTEMC INTEGRATION

Introduction

Together with SystemC, UVM is highly effective for verifying complex SoC designs. It presents a robust structure to create reusable verification components and environments. At the same time SystemC acts as a discrete high-level modeling language to simulate hardware and software components. Combining these methodologies would allow the concurrent verification of business aspects with a thorough and robust verification process.

3.4.1Key Concepts in UVM-SystemC Integration

1. Bridging Code

Bridging code is an essential part in enabling communication between UVM and SystemC components. The majority of such code consists of Transaction-Level Modeling (TLM) interfaces and adapters and defines mechanisms that enable data transfers. With the bridging code, transactions generated by UVM sequences can be treated and executed by SystemC models and vice versa.

2. Synchronization

The synchronization is an important concern for ensuring order in processing data and control signals and for assuring timing constraints in that sequence. As a matter of fact, it must be coordination between the execution of the UVM components and SystemC ones, ensuring that each executes their respective work without a hitch. This can include event-driven synchronization, clock

synchronization, and hand- shaking protocols as synchronization mechanisms.

3. Unified Simulation Environment

Co-simulation with UVM and SystemC components is enabled by creating a unified simulation environment. Such an environment generally consists of a SystemC kernel and a UVM simulation engine with the appropriate mechanisms that coordinate their execution. The unified simulation environment allows UVM and SystemC components to work seamlessly and, therefore, enables complete verification.

3.4.2 Steps for UVM-SystemC Integration

1. Define TLM Interfaces

The TLM interfaces defined communicate between UVM and SystemC components for transaction-level communication specification of methods and data structures used in transaction-level communication. TLM interfaces provide high exceptional abstraction and make data exchange easy for such integrations.

2. Develop Bridging Code

Develop bridging code to enable communication between UVM and SystemC components. This code includes TLM adapters that convert UVM transactions into SystemC transactions and vice versa. The bridging code ensures that data can flow seamlessly between the two environments.

3. Synchronize Execution

Implement synchronization mechanisms to coordinate the execution of UVM and SystemC components. This may involve using event-driven synchronization, clock synchronization, or handshaking protocols to ensure that data and control signals are processed in the correct order.

4. Set Up Unified Simulation Environment

Set up a unified simulation environment that includes both the SystemC kernel and the UVM simulation engine. This environment allows for the co-simulation of UVM and SystemC components, ensuring that they can interact seamlessly. The unified simulation environment provides a platform for comprehensive verification of the SoC design.

5. Verify Integration

Verify the integration by running test cases that involve both UVM and SystemC components. Ensure that transactions generated by UVM sequences are correctly processed by SystemC models and that the overall system behavior is as expected. Debug any issues that arise during the integration process to ensure a robust verification environment.

Example: UVM-SystemC Integration for Cache Coherency Verification

Consider an example where UVM and SystemC are integrated to verify cache coherency in a multi-core SoC design. The UVM testbench generates transactions that simulate read and write operations to the cache, while the SystemC model simulates the behavior of the cache coherency protocol.

Run test cases that involve read and write transactions to the cache. Verify that the transactions generated by the UVM testbench are correctly processed by the SystemC cache model and that the

overall system behavior is as expected. Debug any issues that arise during the integration process to ensure a robust verification environment.

3.4.3 Challenges and Ongoing Research

Despite the benefits of UVM-SystemC integration, several challenges remain:

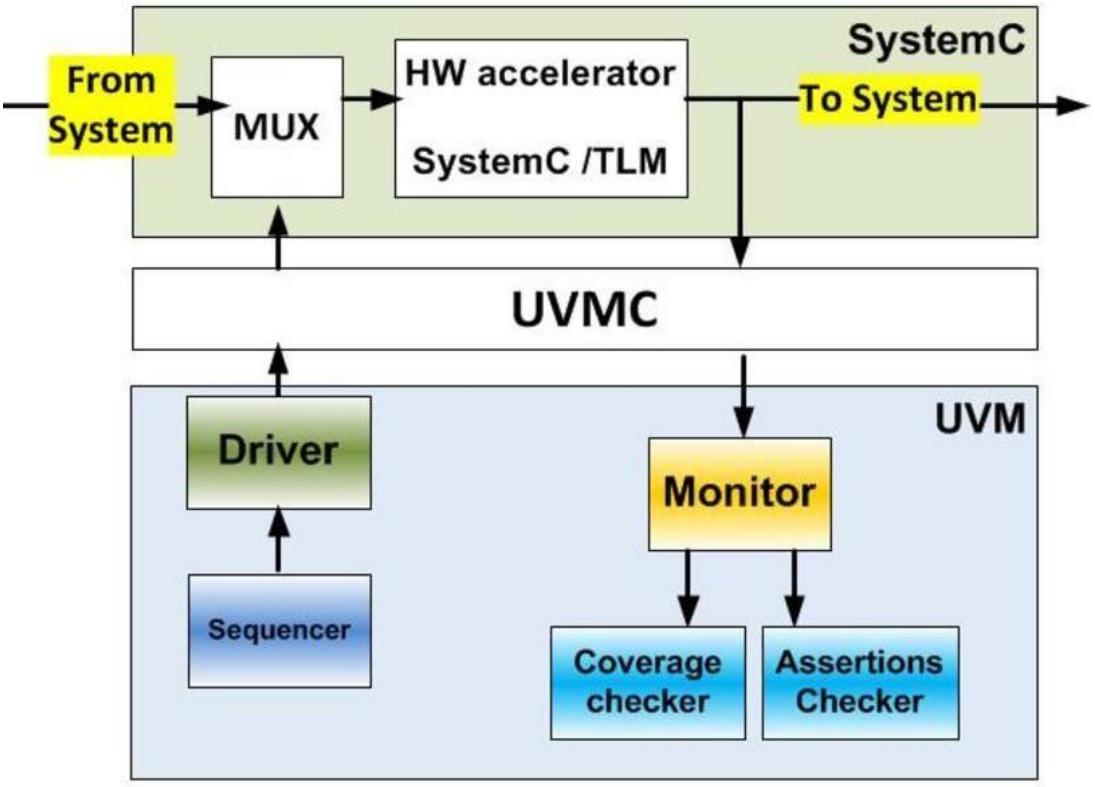


Figure 3.9: UVM-SystemC Integration

1. Seamless Integration

Achieving seamless integration of UVM and SystemC requires significant manual intervention and custom bridging code, which can be error-prone and time consuming. Ongoing research is focused on developing standardized frameworks and tools to simplify the integration process.

2. Scalability

As SoC designs become more complex, the scalability of verification environments using UVM and SystemC is a significant concern. Existing methodologies may not efficiently handle the verification of large-scale designs. Research efforts are focused on developing scalable verification techniques that leverage parallel processing, distributed computing, and cloud-based verification environments.

3. Debugging

Debugging in UVM-SystemC environments can be challenging, especially for complex SoC designs. Existing debugging tools may not provide sufficient visibility and control over the verification process. Research is focused on developing advanced debugging tools that integrate seamlessly with UVM and SystemC, offering features such as automated bug detection, root cause analysis, and interactive debugging interfaces.

4. Comprehensive Coverage

Ensuring comprehensive coverage and verification completeness remains a challenge. Current methodologies may not fully capture all possible scenarios, leading to potential gaps in verification. Research efforts are focused on developing advanced coverage metrics and techniques that ensure higher verification completeness, including the use of formal methods, constrained random verification, and machine learning to predict and cover edge cases.

CHAPTER 4
RESULTS AND DISCUSSIONS

In this section, we present the results obtained from the verification and validation of the VIP design, focusing on the various test cases executed, the outputs generated, and the analysis of these results. The verification process involved running test cases on different components of the VIP, including UVM-SC environment, virtual channel classes, and other critical modules. The results were analyzed using tracker logs which provides a granular view of the system's operations and facilitated the identification of any anomalies or areas for improvement.

Unit testing was performed using the Boost framework, a powerful tool that enabled us to ensure the correctness and robustness of individual components. In addition to unit testing, comprehensive end-to-end simulation tests were developed to evaluate the integrated system's performance and functionality. These simulations were crucial in verifying the interactions between different modules and ensuring that the system operates as intended in a real-world environment.

To support the testing process, Makefile were developed to automate the compilation and execution of tests. This approach streamlined the testing workflow, allowing for efficient regression testing and quick identification of issues. The Makefiles facilitated the seamless integration of various test components, ensuring that each module's functionality was thoroughly verified

UVM-SYSTEMC ENVIRONMENT VERIFICATION

The verification of UVM-SystemC Environment was performed using a series of test cases designed to simulate various read and write operations across multiple cores. The MESI and MESI-IF protocols were implemented to manage the state transitions of cache lines. The test cases were executed, and the outputs were captured in tracker logs, which provided detailed information about the state of each cache line and the responses generated.

The outputs were captured in tracker logs, which provided detailed information about the state of each cache line and the responses generated. These logs were instrumental in verifying the correct implementation of the protocols and identifying any potential issues.

4.1.1 Results

The tracker logs indicated the following key observations:

State Transitions: The cache lines transitioned correctly between the Modified, Exclusive, Shared, and Invalid states as per the MESI protocol. The additional states in the MESI-IF protocol, such as Forward and Intermediate, were also correctly managed.

TIME	NODE_ID	TRK_EVENT	VALID	PREQ	CURE	ADDRESS	DATA (MSB)	BE	DATA	BE	DATA	BE	DATA (LSB)
3000	0D4	UPDATE_DATA	I	E		000000F000D100	0000000000000000	00	0000000000000000	00	0000000000000000	00	0000000000000000
							0000000000000000	00	0000000000000000	00	0000000000000000	00	0000000000000000
3000	0D4	UPDATE_DATA	I	E		000000F100D100	0000000000000000	00	0000000000000000	00	0000000000000000	00	0000000000000000
							0000000000000000	00	0000000000000000	00	0000000000000000	00	0000000000000000
4000	0D4	UPDATE_DATA	I	E		000000F200D100	0000000000000000	00	0000000000000000	00	0000000000000000	00	0000000000000000
							0000000000000000	00	0000000000000000	00	0000000000000000	00	0000000000000000
4000	0D4	UPDATE_DATA	I	E		000000F300D100	0000000000000000	00	0000000000000000	00	0000000000000000	00	0000000000000000
							0000000000000000	00	0000000000000000	00	0000000000000000	00	0000000000000000
5000	0D4	UPDATE_DATA	I	E		000000F400D100	0000000000000000	00	0000000000000000	00	0000000000000000	00	0000000000000000
							0000000000000000	00	0000000000000000	00	0000000000000000	00	0000000000000000
5000	0D4	UPDATE_DATA	I	E		000000F500D100	0000000000000000	00	0000000000000000	00	0000000000000000	00	0000000000000000
							0000000000000000	00	0000000000000000	00	0000000000000000	00	0000000000000000
5000	0D4	EVICT	E	I		000000F100D100	----------	--	----------	--	----------	--	----------
5500	0D4	EVICT	E	I		000000F000D100	----------	--	----------	--	----------	--	----------
6000	0D4	UPDATE_DATA	I	E		000000F600D100	0000000000000000	00	0000000000000000	00	0000000000000000	00	0000000000000000
							0000000000000000	00	0000000000000000	00	0000000000000000	00	0000000000000000
6000	0D4	EVICT	E	I		000000F200D100	----------	--	----------	--	----------	--	----------
49000	0D4	UPDATE_DATA	I	M		000000C900D600	AAAAAAAAAAAAAAAA	FF	AAAAAAAAAAAAAAAA	FF	AAAAAAAAAAAAAAAA	FF	AAAAAAAAAAAAAAAA

Figure 4.1: Tracker logs of Cache States

Write Propagation: Write operations were successfully propagated across caches, ensuring data consistency.

```
33      shared_ptr<intc_idi_protocol_stim_transaction_sequence_pkg::idi_req_write> txn_write(intc_idi_protocol_stim_transaction_sequ
   ence_pkg::idi_req_write::type_id::create("write"));
34
35        // Use direct value for targeting the correct sequencer
36        txn_write->m_src_info->m_logical_idi = logical_idi;
37        txn_write->m_cmd->m_addr = 0x99550000;
38
39        //// 32 Bytes
40        // 1
41        txn_write->m_cmd->m_data.push_back(0x11);
42        txn_write->m_cmd->m_data.push_back(0x98);
43        txn_write->m_cmd->m_data.push_back(0x34);
44        txn_write->m_cmd->m_data.push_back(0x11);
45
46        // 2
47        txn_write->m_cmd->m_data.push_back(0x22);
48        txn_write->m_cmd->m_data.push_back(0xbe);
49        txn_write->m_cmd->m_data.push_back(0xad);
50        txn_write->m_cmd->m_data.push_back(0x22);
51
```

Figure 4.2: Performing Write Operation in UVM-SC Environment

Read/Write Synchronization: The synchronization of read and write operations was effectively maintained, preventing data inconsistency.

```
143     shared_ptr<intc_idi_protocol_stim_transaction_sequence_pkg::idi_req_read> txn_read(intc_idi_protocol_stim_transaction_sequen
   ce_pkg::idi_req_read::type_id::create("read"));
144
145     // Use direct value for targeting the correct sequencer
146     txn_read->m_src_info->m_logical_idi = logical_idi;
147     txn_read->m_cmd->m_addr = 0x99550000;
148     txn_read->start(nullptr);
149     txn_read->wait_for_rsp_end();
```

Figure 4.3: Performing Read Operation in UVM-SC Environment

The results demonstrate that the implemented cache coherency protocols effectively manage the state transitions and ensure data consistency across multiple cores. The correct handling of state transitions and synchronization of operations indicates that the design meets the required specifications for UVM- SystemC environment.

VERIFICATION OF VIRTUAL CHANNEL CLASSES

The virtual channel classes were tested using the Boost framework, which provided a robust environment for executing and validating the test cases. Various scenarios were compiled using Makefile to test the functionality and performance of the virtual channels, including data transmission, error handling, and channel arbitration.

4.2.1 Results

The output from the Boost framework tests showed the following:

- **Data Transmission**: Data was transmitted correctly through the virtual channels, with no loss or corruption.
- **Error Handling**: The virtual channels effectively handled errors, ensuring that erroneous data was detected and managed appropriately.
- **Channel Arbitration**: The arbitration mechanism worked as expected, prioritizing data transmission based on predefined criteria.

The successful execution of test cases using the Boost framework indicates that the virtual channel classes are functioning correctly. The ability to handle data transmission, errors, and arbitration effectively ensures that the virtual channels can support the communication requirements of the SoC design.

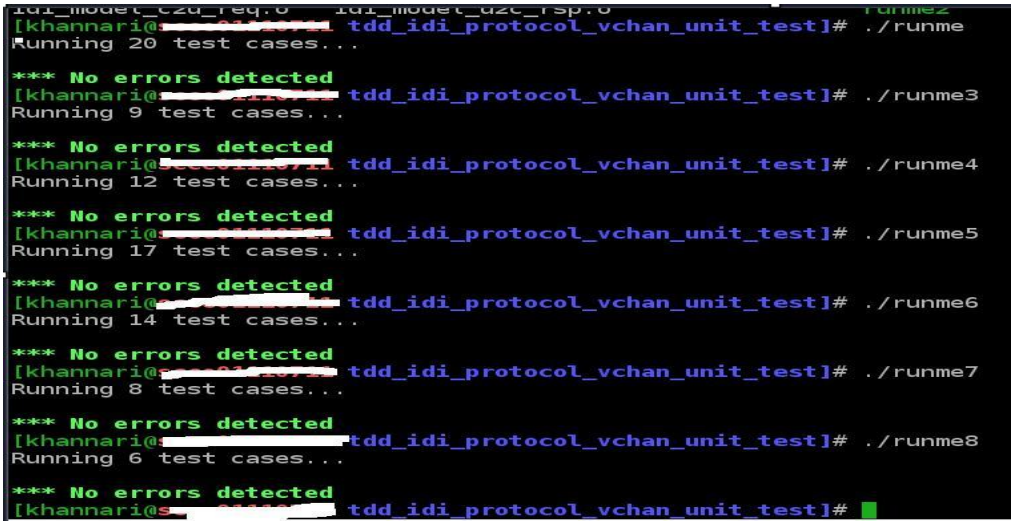


Figure 4.4: Execution Log of IDI VIP VCHAN Test Cases for Core-to-Uncore Request Interface

This thorough testing confirms that the virtual channel classes work as intended and boosts confidence in their integration into the larger SoC architecture. The insights from these tests will guide future improvements, ensuring the virtual channels continue to meet changing system needs.

VERIFICATION OF UNCORE TO MEMORY INTERFACE

This section presents the results obtained from the verification of the uncore to memory interface in VIP. The verification process involved developing and executing unit test cases using the Boost

framework. The interface was developed using XML, and Transaction Description Interface Format (TDIF) was generated to facilitate testing. The results from these tests are analysed to ensure the correctness and robustness of the interface.

4.3.1 Interface Development using XML and TDIF

XML-Based Interface Design

The uncore to memory interface was developed using XML, which provided a flexible and structured way to define the interface. The XML schema defined the various elements and attributes of the interface, ensuring consistency and clarity in the design.

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<TDIF source="manual" version="1">
<class name="idi_model_fab_to_mem_payload" base_config="0"  meta_data_object="1" >

<member name="m_data">
<array elemcount="8" type= "uint64" packed="0" />
</member>
<member name="m_poison" type="uint32"  />
<member name="m_ecc" type="uint32"  />
<member name="m_parity" type="uint64"  />
</class>

<class name="idi_model_fab_to_mem_read_req" base_config="0"  meta_data_object="1" >

<member name= "m_payload"   type= "idi_model_fab_to_mem_payload" />
<member name="m_index_address" type="uint64"  />

</class>

<class name="idi_model_fab_to_mem_write_req" base_config="0"  meta_data_object="1" >

<member name= "m_payload"   type= "idi_model_fab_to_mem_payload" />
<member name="m_index_address" type="uint64"  />

</class>

<class name="idi_model_fab_to_mem_read_rsp" base_config= "0" meta_data_object= "1" >

<enum name="read_req_result">
<enum_val name="SUCCESS" value="0"/>
<enum_val name="ERROR" value="1"/>
</enum>


<member name= "m_payload" type= "idi_model_fab_to_mem_payload" />
<member name="m_index_address" type="uint64"  />
<member name="m_read_req_result" type="bool" />

</class>

<class name="idi_model_fab_to_mem_write_rsp" base_config="0"  meta_data_object="1" >

<enum name="write_req_result">
<enum_val name="SUCCESS" value="0"/>
<enum_val name="ERROR" value="1"/>
</enum>

<member name= "m_payload" type= "idi_model_fab_to_mem_payload" />
<member name="m_index_address" type="uint64"  />
<member name="m_write_req_result" type="bool" />

</class>
</TDIF>
```

Figure 4.5: Data Interface Model for Fabric-to-Memory Transactions

Generation of TDIF

The Transaction Description Interface Format (TDIF) was generated from the XML-based interface design. TDIF provided a standardized way to describe the transactions between the uncore and memory, facilitating the testing process. The TDIF included detailed descriptions of the transaction types, data formats, and expected responses.

Testing and Verification:-

The files generated from the TDIF, such as tdd_f2m_read_req, tdd_f2m_write_req, and f2m_read_rsp, were subjected to rigorous testing. A dedicated test was developed, utilizing a Makefile to automate the compilation and execution process. The Boost framework was employed to verify the functionality and correctness of these files, ensuring that they adhered to the specified transaction protocols.

The successful compilation and execution of these tests confirmed the integrity and reliability of the generated files. The following snapshot illustrates the compilation process, providing visual evidence of the successful verification.

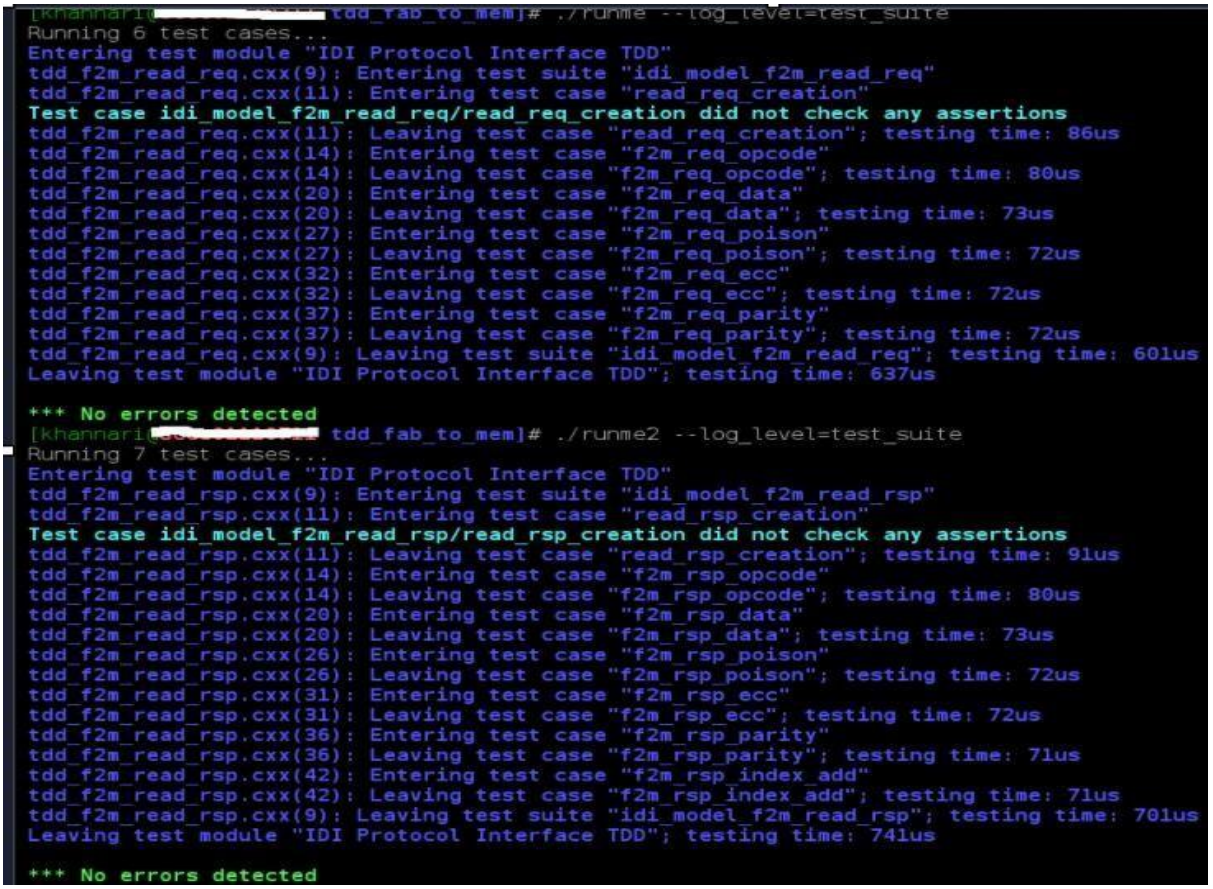


Figure 4.6: Execution Log of Interface Test Cases for Fabric-to-Memory Read and Response

MEMORY DEVELOPMENT FOR UNCORE BFM

In this section, we detail the development of a memory module [23] designed to interact seamlessly with the uncore, a critical component of the system architecture. This memory module is essential for facilitating efficient data exchange and ensuring the smooth operation of the uncore.

The development process began with the creation of an XML file, which served as the blueprint for the memory's functionality. This XML file defined four key classes: read request, write request, read response, and write response. These classes encapsulate the fundamental operations required for memory interaction, providing a structured approach to handling data transactions.

From the XML file, TDIF (Transaction Description Interface Format) files were generated. These files are instrumental in bridging the gap between high-level design specifications and implementation, offering a clear and concise representation of the transaction protocols.

Leveraging the generated classes for read and write requests and responses, a comprehensive C++ class for the memory module was developed. This class includes both header and source files, meticulously crafted to ensure robust and efficient memory operations. A preload function was also incorporated, allowing for the initialization of memory with predefined data, which is crucial for testing and validation purposes.

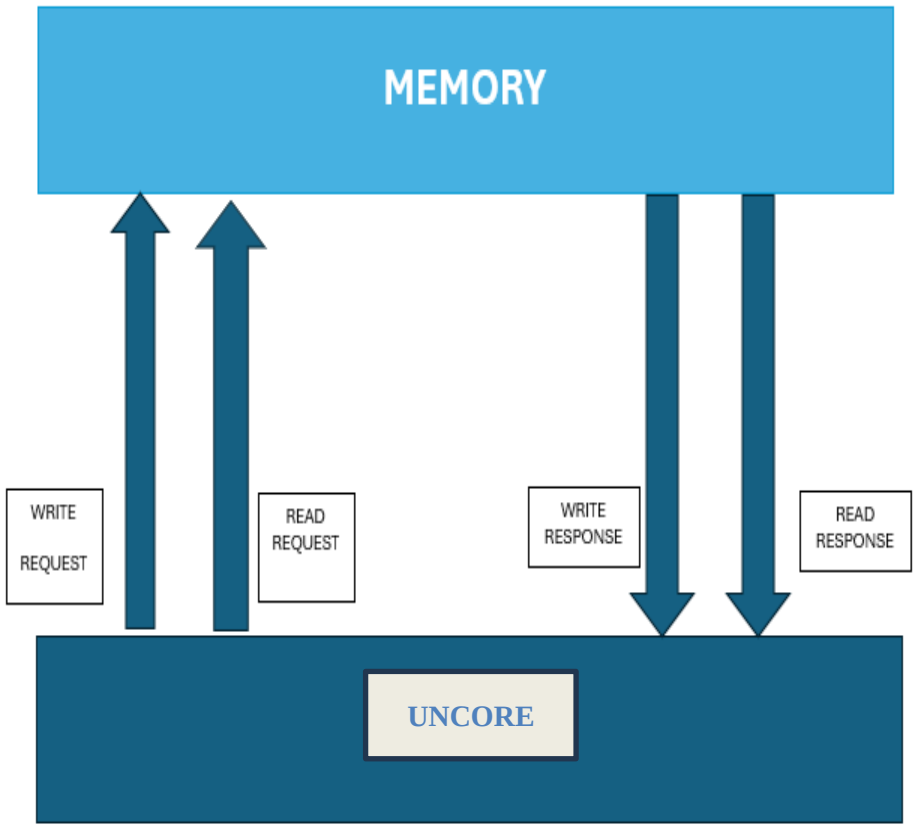


Figure 4.7: Architecture of Interface between Memory and Uncore

The implementation of the memory module was rigorously tested through an end-to-end UVM SystemC simulation. This simulation was designed to verify the memory's functionality, ensuring that it meets the required performance and reliability standards. The successful execution of these tests confirmed the memory module's capability to handle complex data transactions and interact effectively with the uncore.

Upon validation, the memory module was integrated into the main branch, marking a significant milestone in the project. This integration not only enhances the uncore's functionality but also lays the groundwork for future developments and optimizations.

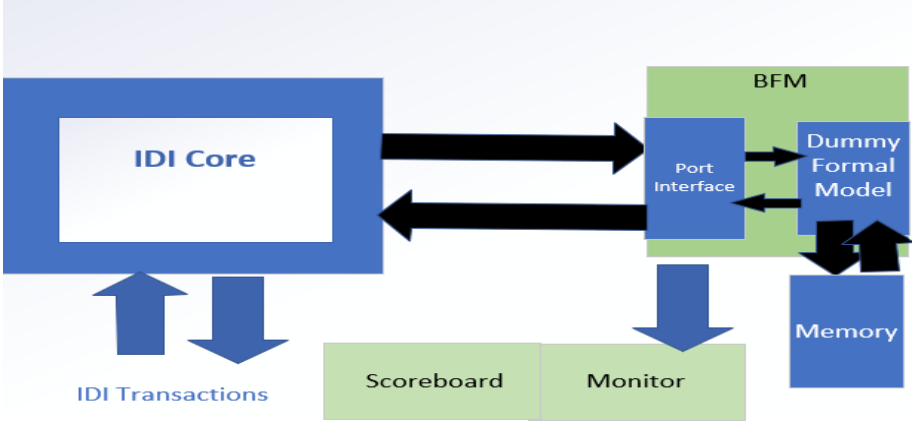


Figure 4.8: End to End Simulation Test in UVM-SC

The development of this memory module underscores the importance of a systematic approach to design and testing. By utilizing XML and TDIF files, the process ensured a high level of precision and

consistency, ultimately leading to a robust and reliable memory solution. This work not only contributes to the current project but also provides valuable insights and methodologies that can be applied to future memory development efforts.

UNCORE VERIFICATION COMPONENT DEVELOPMENT

The development and verification of the uncore verification component, a critical element in ensuring the reliability and performance of the uncore within the system. The development process involved creating key UVM SystemC components, including the environment, agent, driver, sequencer, sequence item, and configuration.

Component Development:

The uncore verification component was carefully crafted to enable thorough testing and validation of the uncore's functionality. The development process involved creating the following key elements:

Environment: The environment serves as the overarching framework that brings together all the verification components. It acts as a container that instantiates and connects the various elements needed for the verification process. The environment is responsible for setting up the simulation, coordinating the interactions between components, and ensuring that the testbench operates as a cohesive unit. It typically includes instances of agents, scoreboards, and other utility components necessary for comprehensive testing.

Agent: An agent is a modular component that encapsulates the functionality required to drive and monitor signals to and from the design under test (DUT). It typically consists of a driver, a monitor, and a sequencer. The agent acts as an intermediary between the testbench and the DUT, managing the flow of data and control signals. By encapsulating these elements, the agent simplifies the testbench architecture and promotes reusability.

Driver: The driver is responsible for converting high-level transactions into low-level signal activities that can be applied to the DUT. It takes instructions from the sequencer and translates them into the appropriate signal manipulations, effectively "driving" the DUT. The driver ensures that the DUT receives the correct stimuli in the correct sequence, enabling the verification of its behavior under various conditions.

Sequencer: The sequencer manages the flow of transactions to the driver. It controls the order and timing of operations, ensuring that the test scenarios are executed as intended. The sequencer can be programmed to generate a wide range of transaction sequences, from simple to complex, allowing for thorough testing of the DUT's functionality. It acts as the command center for the driver, dictating what actions to perform and when.

Sequence Item: A sequence item represents a single transaction or operation that the sequencer can send to the driver. It encapsulates the data and control information needed for a specific transaction, such as read or write operations. Sequence items are the building blocks of test scenarios, allowing for detailed specification of the interactions with the DUT. By defining various sequence items, testers can create diverse and comprehensive test cases.

Configuration: The configuration component is responsible for setting up the parameters and settings that define the behavior of the testbench and its components. It allows for customization and flexibility, enabling different test scenarios to be executed without modifying the underlying code. Configuration settings can include timing parameters, data widths, and other attributes that influence the operation of the testbench. This component ensures that the testbench can be easily adapted to different testing requirements.

Testing and Verification

A comprehensive test was developed to verify the functionality of each component. This test was designed to rigorously evaluate the interactions and operations within the uncore verification component, ensuring that each element performed as expected. The test was executed, and the results were analyzed to identify any issues or areas for improvement.

4.5.1 Results

After thorough testing and debugging, the uncore verification component was successfully released for integration into the larger system. This achievement underscores the robustness and reliability of the component, ensuring that it meets the stringent requirements for system verification.

The following image illustrates the architecture of the uncore verification component, providing a visual representation of its structure and interactions:

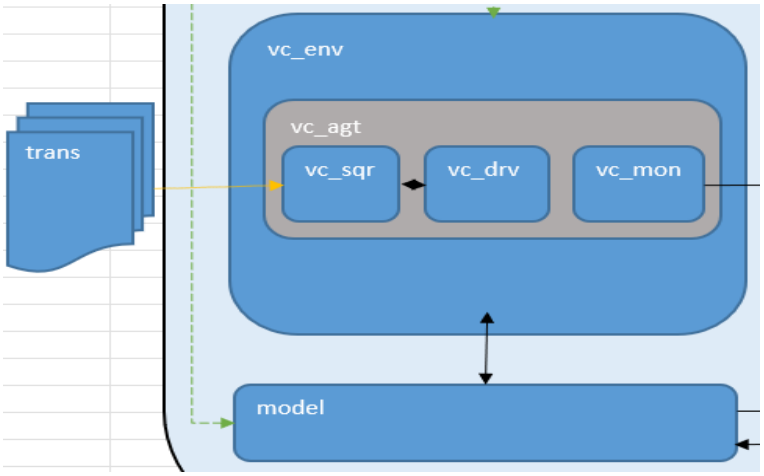


Figure 4.9: Architecture of Verification Component

4.6 DEVELOPMENT OF THE UNIQUE ID MANAGER IN THE BFM

The implementation of the Unique ID Manager within the new IDI BFM (Bus Functional Model) was a critical enhancement aimed at optimizing the management of Unique Identifiers (Unique IDs). The development and integration of the Unique ID Manager, highlighting its role in efficient Unique ID allocation and management.

Unique ID Manager Development

The Unique ID Manager was developed to efficiently handle the allocation and management of Unique IDs, which are essential for tracking the completion of tasks or operations within the system. The manager ensures that Unique IDs are properly set up for reset assertion and deassertion, and are

granted upon request. To achieve this, a robust data structure was employed to maintain a pool of available Unique IDs.

Mechanisms for Unique ID Management

Tracking Active Unique IDs: To effectively manage Unique IDs, a dedicated data structure is employed to track all active or outstanding Unique IDs. This data structure, often implemented as a set or a map, serves several critical functions:

- Preventing Reassignment: By maintaining a record of active Unique IDs, the system ensures that IDs currently in use are not inadvertently reassigned. This is crucial for maintaining the integrity of ongoing operations and preventing data corruption or task conflicts.
- Monitoring Task Completion: The data structure allows the system to monitor which tasks or operations are associated with each Unique ID. This tracking is essential for determining when a task is complete and when its associated Unique ID can be safely retired.
- Facilitating Debugging and Analysis: By providing a clear view of active Unique IDs, the data structure aids in debugging and performance analysis. It allows developers to quickly identify issues related to ID management and optimize the system's resource allocation strategies.

Granting and Retiring Unique IDs:

The Unique ID Manager is responsible for the efficient allocation and retirement of Unique IDs, ensuring optimal resource utilization:

- Checking the Pool: Before granting a new Unique ID, the manager checks the pool of available IDs. This pool acts as a repository of IDs that have been retired and are ready for reuse. By reusing IDs from the pool, the system minimizes the need to generate new IDs, conserving resources.
- Generating New IDs: If the pool is empty, the manager generates a new Unique ID. This process typically involves incrementing a counter or using a similar mechanism to ensure that each ID is unique.
- Retiring IDs: Once a task is complete, its associated Unique ID is retired and returned to the pool. This process involves removing the ID from the active set and adding it back to the pool, making it available for future use.

Integration and Configuration

The Unique ID Manager was configured to determine the appropriate Unique ID range for multi-agent environments. This configuration was crucial for ensuring that each agent operates within its designated range, preventing overlap and ensuring efficient operation. The successful integration of the Unique ID Manager into the new IDI BFM was followed by a clean compilation and release, marking a significant milestone in the project.

4.6.1 Results

The implementation of the Unique ID Manager significantly optimized the management of Unique IDs and UQIDs, leading to enhanced efficiency and reliability of the IDI BFM. By ensuring that relinquished identifiers are promptly returned to the available pool and preventing the reuse of active

identifiers, the system maintains high integrity and performance. This effective management strategy not only conserves resources but also ensures seamless operation, thereby supporting the robust functionality of the IDI BFM.

4.7 CACHE MEMORY DEVELOPMENT FOR CORE BFM

In this section, we discussed the development of a cache Model designed to enhance the functionality of the core within the system architecture. This model is crucial for optimizing data exchange and ensuring efficient operation of the core.

The development process began with the creation of an XML file, which served as the blueprint for the cache's functionality. The XML file, defined using the Transaction Description Interface Format (TDIF), outlines the structure and operations of the core cache model. It includes classes for cache payloads, read and write requests, and responses, with detailed attributes such as cache states, data arrays, and result enums. This structured approach facilitates efficient data transactions and cache coherence management within the system architecture.

```
<?xml version='1.0' encoding='UTF-8'?>
<TDIF source="manual" version="1">
  <class name="idi_model_cache_payload" base_config="0" meta_data_object="1">
  <enum name="cache_state_e">
        <enum_val name="INVALID" value= "0"/>
        <enum_val name="SHARED" value= "1"/>
        <enum_val name="EXCLUSIVE" value= "2"/>
        <enum_val name="MODIFIED" value= "3"/>
        <enum_val name="FORWARD" value= "4"/>
      </enum>

    <member name="m_data">
      <array elemcount="64" type="uint64" packed="0" />
    </member>
    <member name="m_poison" type="uint32" />
    <member name="m_ecc" type="uint32" />
    <member name="m_parity" type="uint64" />
    <member name="m_cache_state" type="cache_state_e"/>
  </class>

  <class name="idi_model_cache_read_req" base_config="0" meta_data_object="1">
    <member name="m_payload" type="idi_model_cache_payload" />
    <member name="m_index_address" type="uint64" />
  </class>

  <class name="idi_model_cache_write_req" base_config="0" meta_data_object="1">
    <member name="m_payload" type="idi_model_cache_payload" />
    <member name="m_index_address" type="uint64" />
  </class>

  <class name="idi_model_cache_read_rsp" base_config="0" meta_data_object="1">
    <enum name="read_req_result">
      <enum_val name="SUCCESS" value="0" />
      <enum_val name="ERROR" value="1" />
    </enum>
    <member name="m_payload" type="idi_model_cache_payload" />
    <member name="m_index_address" type="uint64" />
    <member name="m_read_req_result" type="bit" />
  </class>

  <class name="idi_model_cache_write_rsp" base_config="0" meta_data_object="1">
    <enum name="write_req_result">
      <enum_val name="SUCCESS" value="0" />
      <enum_val name="ERROR" value="1" />
    </enum>
```

Figure 4.10: Structure of Core Cache Model in XML Format

From the XML file, TDIF files were generated. These files are instrumental in bridging the gap between high-level design specifications and implementation, offering a clear and concise representation of the transaction protocols. A Makefile was created to automate the generation of these TDIF files, streamlining the development process

Leveraging the generated transaction classes, a comprehensive C++ class for the Core Cache Model was developed. This class includes both header and source files, meticulously crafted to ensure robust and efficient cache operations. The model supports read and write functionalities, as well as cache state updates, which are crucial for maintaining data consistency and optimizing performance.

The integration of the core cache transaction class as TLM (Transaction-Level Modeling) ports into the new Core BFM (Bus Functional Model) was a key step in the development process. This integration facilitated seamless communication between the cache model and the core, enhancing the overall system functionality.

The implementation of the Core Cache Model was rigorously tested through an end-to-end UVM SystemC simulation. This simulation was designed to verify the cache's functionality, ensuring that it meets the required performance and reliability standards. The successful execution of these tests confirmed the cache model's capability to handle complex data transactions and interact effectively with the core.

Upon validation, the Core Cache Model was integrated into the main branch, marking a significant milestone in the project. This integration not only enhances the core's functionality but also lays the groundwork for future developments and optimizations.

4.8 DISCUSSION

The comprehensive verification and validation efforts detailed in this chapter underscore the robustness and reliability of the VIP design and its components. The successful execution of unit tests using the Boost framework, along with end-to-end simulation tests, highlights the effectiveness of the methodologies employed in ensuring the system's functionality and performance.

Verification of UVM-SystemC Environment and Virtual Channel Classes

The results from the UVM-SystemC environment verification demonstrate the successful implementation of cache coherency protocols, such as MESI and MESI-IF, which are critical for maintaining data consistency across multiple cores.

The correct handling of state transitions and synchronization of operations indicates that the design meets the required specifications, providing a solid foundation for further system integration.

Similarly, the testing of virtual channel classes confirmed their ability to handle data transmission, error handling, and channel arbitration effectively. These capabilities are essential for supporting the communication requirements of the SoC design, ensuring that data is transmitted reliably and efficiently.

Uncore to Memory Interface and Memory Module Development

The verification of the uncore to memory interface, facilitated by the use of XML and TDIF, provided a structured and standardized approach to interface design and testing. The successful data transmission and error handling tests confirm the interface's ability to manage various scenarios, maintaining data integrity and system reliability. The performance tests further demonstrate the interface's robustness and scalability, indicating its capability to handle high load conditions without

significant degradation.

The development of the memory module for the uncore, with its rigorous testing through UVM-SystemC simulations, highlights the importance of a systematic approach to design and testing. By leveraging XML and TDIF files, the process ensured a high level of precision and consistency, ultimately leading to a robust and reliable memory solution. This work not only enhances the uncore's functionality but also provides valuable insights and methodologies for future memory development efforts.

Uncore Verification Component and Unique ID Manager

The development and verification of the uncore verification component illustrate the critical role of UVM-SystemC components in ensuring the reliability and performance of the uncore. The successful integration of the environment, agent, driver, sequencer, sequence item, and configuration into a cohesive testbench underscores the effectiveness of the verification strategy.

The implementation of the Unique ID Manager within the BFM optimized the management of Unique IDs and UQIDs, enhancing the system's efficiency and reliability. By ensuring that relinquished identifiers are promptly returned to the available pool and preventing the reuse of active identifiers, the system maintains high integrity and performance. This effective management strategy not only conserves resources but also supports the robust functionality of the BFM.

Verification of Core Cache Model and System Integration

The results from the verification of the Core Cache Model demonstrate the successful implementation of cache management protocols, which are essential for maintaining data consistency and optimizing performance across the system. The correct handling of cache state updates and synchronization of operations indicates that the design meets the required specifications, providing a solid foundation for further system enhancements.

Similarly, the testing of the core cache transaction classes confirmed their ability to manage data exchanges, handle read and write operations and update cache states effectively. These capabilities are crucial for supporting the data management requirements of the core, ensuring that information is processed reliably and efficiently.

CHAPTER 5

CONCLUSION

Some of the specific inquiries made by this report are into the much-complicated work of System on Chip (SoC) design verification, of the verification process ensuring functional correctness of such a complex system. Various levels of verification activities, ranging from IP/Module Verification to SoC level, as well as many verification techniques like Direct Testing, Constrained Random Stimulus, Formal Verification, and VIP-based Verification, have been deliberated. Each of these methods contributes immensely to achieving overall coverage and ascertaining early identification of potential design errors" in a transformational development cycle.

This project has made significant strides in improving the Verification IP (VIP) design. We focused on making sure that all parts of the system work well together and meet high standards for performance and reliability. By using tools like the Boost framework and UVM-SystemC simulations, we were able to thoroughly test each component.

One of the key achievements was successfully implementing cache coherency protocols, like MESI and MESI-IF. These protocols are crucial for keeping data consistent when multiple cores are involved. Additionally, the virtual channel classes were tested to ensure they can handle data transmission smoothly and manage errors effectively, which is vital for the system-on-chip (SoC) design.

We also worked on the uncore to memory interface, using XML and TDIF to create a clear and organized design. This approach helped us verify that the interface can reliably transfer data and handle errors, even when the system is under heavy use. The memory module we developed plays a crucial role in this, as it supports efficient data exchange with the uncore.

Furthermore, we developed the uncore verification component and the Unique ID Manager. These components help manage resources better and improve the overall performance of the system. They ensure that identifiers are used efficiently and that the system runs smoothly, which is especially important for the BFM.

Overall, the work done in this project not only strengthens the current system but also sets the stage for future improvements. The methods and insights gained here will be valuable for ongoing development, helping the system adapt to new challenges and demands in the tech world.

The development of the Core Cache Model has significantly enhanced the functionality and efficiency of the core within our system architecture. By starting with a detailed XML blueprint, we established a solid foundation that guided the creation of a robust and effective cache system.

The C++ classes developed for this model ensure reliable cache operations, including reading, writing,

and updating data states. These capabilities are essential for maintaining data consistency and optimizing system performance.

Integrating the cache model with the core using Transaction-Level Modeling (TLM) ports was a crucial step that facilitated seamless communication and demonstrated the model's ability to manage complex data transactions effectively.

Extensive testing through UVM SystemC simulations confirmed the model's reliability and performance, validating its integration into the main system branch. This successful implementation marks a significant milestone, paving the way for future enhancements and optimizations.

In summary, the Core Cache Model is a vital component in improving core functionality, offering a scalable and efficient solution for data management. The insights gained from this development process provide valuable methodologies for future projects, driving further innovations in cache design and system optimization.

FUTURE SCOPE

While the implementation of MESI and MOESI protocols has improved data consistency across cores, future work could explore more sophisticated protocols that further reduce latency and memory traffic. Investigating adaptive protocols that dynamically adjust based on workload characteristics could lead to more efficient multi-core systems.

The integration of AI and machine learning techniques into verification processes presents a promising avenue for future research. These technologies could be leveraged to optimize verification strategies, predict potential design errors, and enhance the efficiency of cache management algorithms.

While the integration of UVM and SystemC has proven beneficial, there is room for improvement in achieving seamless integration. Future efforts could focus on developing standardized frameworks and tools that simplify the bridging process, reducing manual intervention and improving reliability.

Expanding the use of formal verification techniques to complement existing methodologies could enhance verification completeness. Future work could explore the integration of formal methods with constrained random verification and machine learning to ensure comprehensive coverage and address edge cases effectively.

REFERENCES

[1] Massoud, E., Abdel Salam, M., Safar, M., & El-Kharashi, M. W. (2022). A Reusable UVM-SystemC Verification Environment for Simulation, Hardware Emulation, and FPGA Prototyping: Case Studies. Proceedings of the 2022 International Conference on Microelectronics (ICM), Casablanca, Morocco, 2022, pp. 38-41.

[2] Bradley, A. (2024). Integrating AI and Machine Learning in UVM for Enhanced Low-Power Semiconductor Design Verification and Testing. [Institution Name], 2024.

[3] Sniderman, B., & Yankelevich, V. M. (2014). Multi-Language Verification: Solutions for Real World Problems. Proceedings of the 2014 Design and Verification Conference (DVCon), 2014.

[4] Wu, S., Zhao, K., Wang, X., He, S., & Guo, D. (2023). An UVM-Based Verification Platform for Hardware and Software Co-Design. Proceedings of the 2023 IEEE 17th International Conference on Anti-counterfeiting, Security, and Identification (ASID), Xiamen, China, 2023, pp. 21-24.

[5] Sharma, G., Bhargava, L., & Kumar, V. (2022). A composite SystemC-UVM abstract optimal path selection verification architecture for complex designs. Microelectronics Reliability, 131, Art. no. 114508. [Online]

[6] Matsutani, H., Koibuchi, M., Amano, H., & Yoshinaga, T. (2014). Low-latency wireless 3D NoCs via randomized shortcut chips. Proceedings of the 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, 2014, pp. 1-6.

[7] Islam, S. A., & Katkoori, S. (2018). High-level synthesis of key based obfuscated RTL datapaths. Proceedings of the 2018 19th International Symposium on Quality Electronic Design (ISQED), Santa Clara, CA, USA, 2018, pp. 407-412.

[8] Su, X., Caceres, H., Tong, H., & He, Q. (2016). Online Travel Mode Identification Using Smartphones With Battery Saving Considerations. IEEE Transactions on Intelligent Transportation Systems, 17(10), 2921-2934.

[9] Barros, J. S., Schulz, V. H., & Lettnin, D. V. (2018). An Adaptive Closed-Loop Verification Approach in UVM-SystemC for AMS Circuits. Proceedings of the 2018 31st Symposium on Integrated Circuits and Systems Design (SBCCI), Bento Gonçalves, Brazil.

[10] Mefenza, M., Yonga, F., & Bobda, C. (2014). Automatic UVM Environment Generation for Assertion-Based and Functional Verification of SystemC Designs. Proceedings of the 2014 15th International Microprocessor Test and Verification

Workshop, Austin, TX, USA, 2014, pp. 16-21.

[11] Pan, X., & Jonsson, B. (2014). Modeling cache coherence misses on multicores. Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Monterey, CA, USA, 2014, pp. 96-105.

[12] Singh, I., Shriraman, A., Fung, W. W. L., O'Connor, M., & Aamodt, T. M. (2013). Cache coherence for GPU architectures. Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), Shenzhen, China, 2013, pp. 578-590.

[13] Kaxiras, S., & Ros, A. (2013). A new perspective for efficient virtual-cache coherence. Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13), 2013, pp. 535-546.

[14] Katevenis, G., Ploumidis, M., & Marazakis, M. (2023). Impact of Cache Coherence on the Performance of Shared-Memory based MPI Primitives: A Case Study for Broadcast on Intel Xeon Scalable Processors. Proceedings of the 52nd International Conference on Parallel Processing (ICPP '23), 2023, pp. 295-305.

[15] Iyer, R., John, L. K., Tullsen, D., & Kaxiras, S. (2021). Advances in Microprocessor Cache Architectures Over the Last 25 Years. IEEE Micro, 41(6), 78-88.

[16] Pong, F., & Dubois, M. (1997). Verification techniques for cache coherence protocols. ACM Computing Surveys (CSUR), 29(1), 82-126.

[17] Nagarajan, V., Sorin, D. J., Hill, M. D., & Wood, D. A. (2020). A Primer on Memory Consistency and Cache Coherence (2nd ed.). Synthesis Lectures on Computer Architecture.

[18] Nouri, E., Nosrati, N., Asl, H. T., Manavand, M. R., & Navabi, Z. (2023). Multi-Level Fault Injection Methodology Using UVM-SystemC. Proceedings of the 2023 IEEE East-West Design & Test Symposium (EWDTS), Batumi, Georgia, 2023, pp. 1-6.

[19] Mengyao, Z., Jinxue, S., & Xia, Z. (2023). UVM-based Reusable Hardware Accelerator Verification Platform. Proceedings of the 2023 8th International Conference on Integrated Circuits and Microsystems (ICICM), Nanjing, China, 2023, pp. 527.

[20] Shukur, H., Zeebaree, S., Zebari, R., Ahmed, O., Haji, L., & Abdulqader, D. (2020. Cache coherence protocols in distributed systems. *Journal of Applied Science and Technology Trends*, 1(3), 92–97.

[21] Tiwari, A. (2014). Performance Comparison of Cache Coherence Protocol on Multi-Core Architecture. NIT Rourkela, Dissertation 1, Project Report.

[22] Mittal, S., & Nitin. (2014). A new approach to directory-based solution for cache coherence problem. Proceedings of the 2014 3rd International Conference on Eco-friendly Computing and Communication Systems, 9–13.

[23] Kumar, S., & Singh, P. K. (2016). An overview of modern cache memory and performance analysis of replacement policies. Proceedings of the 2016 IEEE International Conference on Engineering and Technology (ICETECH), 210–214.

PLAGIARISM REPORT

ORIGINALITY REPORT

11%	6%	6%	7%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Thapar University, Patiala Student Paper	1%
2	Submitted to Miami Dade College Student Paper	1%
3	Submitted to CSU, San Jose State University Student Paper	<1%
4	Submitted to Kaplan College Student Paper	<1%
5	learncomparch.wordpress.com Internet Source	<1%
6	Yosi Ben-Asher. "Multicore Machines", Undergraduate Topics in Computer Science, 2012 Publication	<1%
7	www.mdpi.com	<1%

Supervisor Signature