# Multi Core Cache Coherence

**Submitted by:**

2021-FYP-02

| | |
|---|---|
| Abeera Adnan | 2021-EE-106 |
| Ammar Saqib | 2021-EE-141 |
| Muhammad Zeeshan Malik | 2021-EE-143 |

**Supervised by:** Mr. Umer Shahid & Dr. Muhammad Tahir

Department of Electrical Engineering

**University of Engineering and Technology Lahore**

# Multi Core Cache Coherence

Submitted to the faculty of Electrical Engineering
of the University of Engineering and Technology Lahore
in partial fulfillment of the requirements for the Degree of

Bachelor of Science

in

**Electrical Engineering**.

| | |
|---|---|
| Internal Examiner | External Examiner |

Director
Undergraduate Studies

Department of Electrical Engineering

**University of Engineering and Technology Lahore**

# Declaration

I declare that the work contained in this thesis is my own, except where explicitly stated otherwise. In addition this work has not been submitted to obtain another degree or professional qualification.

Signed: ____________________

Date: ____________________

# Acknowledgments

We would like to express our sincere gratitude to our supervisor, **Dr. Umer Shahid**, for his invaluable guidance, continuous support, and insightful feedback throughout our research on Quad-Core Cache Coherence System using UVM Verification. His expertise and encouragement have been instrumental in shaping this work. We also extend our appreciation to our co-supervisor, **Dr. Muhammad Tahir**, for his valuable suggestions and technical insights, which greatly contributed to the progress of this research. Finally, we are thankful to the **University of Engineering and Technology (UET)** for providing the necessary resources and an enriching academic environment that facilitated this research. We also thank our loving **parents** for their support in difficult times, whose encouragement and guidance have been a constant source of strength during challenging times

.

*Dedicated to my groupmates, project advisor, parents, teachers, and the Department of Electrical Engineering, UET Lahore.*

# Contribution to Sustainable Development Goals

Among the provided list of sustainable goals, our project can contribute to the following:

## 0.1 Goal 8: Decent Work and Economic Growth

This project encourages the advancement of efficient, energy-saving computing systems that are crucial for contemporary digital economies. Through the use of advanced design and verification techniques like UVM, the study contributes to generating skilled engineering employment and promotes innovation within the semiconductor and embedded systems sectors. These inputs facilitate economic development by means of technological progress and workforce preparedness.

## 0.2 Goal 9: Industry, Innovation and Infrastructure

The core focus of this thesis is the development of a scalable cache coherence controller utilizing the MESI protocol. These technologies are essential for establishing dependable and efficient computing infrastructure. The study promotes creativity in system-on-chip (SoC) design and backs industrial uses that rely on strong processing systems, establishing the groundwork for more intelligent, interconnected infrastructure.

## 0.3 Goal 7: Affordable and Clean Energy

Although not directly focused on energy systems, the project indirectly aids this objective by creating low-latency and energy-efficient interconnects and coherence methods. Minimizing the communication overhead between processors and memory reduces total system power usage, which is vital for energy-efficient computing platforms in extensive data centers and devices with limited battery capacity at the edge.

## 0.4 Goal 13: Climate Action

This project aids in lowering the environmental impact of computing systems by reducing energy usage in digital hardware through effective cache coherence methods. It aids climate action initiatives by promoting eco-friendly technologies, lowering carbon emissions from energy-demanding computing, and fostering sustainable design in hardware creation.

## 0.5 Goal 4: Quality Education

The project incorporates formal verification methods and industry standards such as UVM, improving the academic integrity and practical significance of engineering education. It provides students with practical experience in advanced hardware design while developing the problem-solving and critical thinking skills essential for careers in embedded systems and digital design.

## 0.6 Goal 11: Sustainable Cities and Communities

This project contributes to the advancement of sustainable urban infrastructure by developing an efficient and coherent multi-core processing system using a MESI-based cache coherence protocol. Such architectures are foundational for smart city applications that demand high-performance and low-power embedded systems — including intelligent traffic control, urban surveillance, environmental monitoring, and connected healthcare devices. By ensuring reliable data consistency and minimizing energy consumption across processor cores, the design supports scalable and energy-aware processing in edge and IoT platforms, ultimately promoting the long-term sustainability of digital infrastructure in smart cities.

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| **HDL** | **H**ardware **D**escription **L**anguage |
| **RTL** | **R**egister **T**ransfer **L**evel |
| **UVM** | **U**niversal **V**erification **M**ethodology |
| **MESI** | **M**odified, **E**xclusive, **S**hared, and **I**nvalid Protocol |
| **SV** | **S**ystem**V**erilog |
| **CCU** | **C**ache **C**ontroller **U**nit |
| **FSM** | **F**inite **S**tate **M**achine |
| **LLC** | **L**ast-**L**evel **C**ache |
| **FV** | **F**ormal **V**erification |
| **PLRU** | **P**seudo-**L**east **R**ecently **U**sed |

# Abstract

This thesis presents the design and verification of a quad-core L1 cache coherence system based on the snooping MESI (Modified, Exclusive, Shared, Invalid) protocol. The system ensures cache coherence in a shared-memory multicore environment while addressing key challenges such as transient states, race conditions, and arbitration fairness. Universal Verification Methodology (UVM) is employed to simulate and validate the architecture, ensuring correctness under diverse inter-core interaction scenarios. The verification process focuses on detecting coherence violations, validating bus arbitration fairness, and testing the functionality of critical components, including the L1 cache and arbiter. Simulation results confirm the system's ability to maintain coherence and fairness under varying conditions, offering a robust framework for future multicore cache designs.

# Chapter 1

# Introduction

Multi-core designs aim to execute several threads or processes in parallel; each core typically comes with its own private cache to prevent latency in the common access of frequently used data [1]. For high performance and scalability, efficient use of memory is very important in modern computing systems, particularly in multi-core architectures[3]. But when multiple cores share a space space and access same data, addressing cache coherence becomes essential to avoid inconsistencies and ensure reliable performance..In other words, cache coherence ensures a consistent view of shared memory across all cores, regardless of the fact that each core may have its local copy of data inside its cache. When cores are running different versions of outdated or wrong data, inconsistencies do arise in computations. Its behavior and thus computations become erratic and erroneous with the increase in the number of cores, which brings about the significant problem of cache coherence in multi-core systems [4].A very common solution to maintaining cache coherence in multi-core processors is the MESI protocol, named after its four states. It assigns one of these four states to each line or block of memory in a core's private cache:

**Modified**: The cache contains a unique copy of the data and has the modified data of the main memory.
**Exclusive**: The cache contains a unique copy of the data, but identical to the main memory.
**Shared**: The data resides in several caches and same as the main memory.
**Invalid** : Data is stale, and a copy of the data has to be fetched from another core or from memory.
While it is effective in solving cache coherence at the architectural level, the point of hardware implementation imposes new difficulties on translations from MESI protocol. Increasing core counts, split-transaction buses, and aggressive performance optimizations create difficulties with which processor designs are becoming increasingly complex, so too do they make the challenge of implementing cache coherence efficiently and correctly.The overall problem of implementing it in hardware arises primarily because of non-atomic transitions in the state of the cache. State transitions in the MESI protocol are treated

as atomic in the architectural model-that is, they constitute a single indivisible step-but in real hardware, every transition of state entails a few instructions, and several cores may simultaneously be trying to access or change the same cache line. This generates several problems:In the multi-core system, cores are working parallel to each other, and their actions interfere with each other's.

**Race Conditions:** For example, consider a scenario where two cores simultaneously attempt to write to the same memory location. Without proper coordination, such cases can lead to race conditions, causing data inconsistency or corruption. Therefore, careful implementation of the MESI protocol is essential to ensure that such conflicts are resolved deterministically, maintaining coherence across cores.

**Transient States:** In hardware implementations of coherence protocols, transitions between states—such as from *Shared* to *Exclusive*—do not occur instantaneously. Instead, the system may temporarily reside in intermediate or transient states. During these periods, the data might not be fully consistent, and it is critical to prevent any core from accessing stale or invalid data. Proper management of these transient states is crucial to ensure the stability and correctness of the coherence mechanism.

**Split-Transaction Buses:** Modern multicore processors often employ split-transaction buses to increase bus utilization by allowing multiple transactions to proceed concurrently. While this improves performance, it significantly increases the complexity of the coherence protocol. The system must now manage additional transient states and potential timing-related issues, as memory operations may overlap. The MESI protocol must be carefully designed to maintain correctness and avoid coherence violations in such environments.

**Verification Complexity**: Verifying the correctness of a hardware implementation of a cache coherence protocol, such as MESI, is inherently complex. While simulation techniques can validate specific scenarios, they are insufficient for guaranteeing correctness across all possible execution paths, particularly with respect to issues such as deadlock, livelock, or starvation. Formal verification techniques, though more comprehensive, are often impractical for complex multicore systems due to state-space explosion. In our quad-core system, the challenge lies in managing concurrent and non-atomic state transitions, which can give rise to transient states, race conditions, and potential deadlocks. These challenges are further intensified by the use of split-transaction buses and priority-based arbitration, which, while improving bus access efficiency, introduce additional complexity in ensuring fair and correct coherence behavior. As a result, maintaining coherence, avoiding starvation, and verifying correctness under all conditions requires a carefully structured design and verification strategy.

# Chapter 2

# Problem Statement

Our FYDP (Final Year Design Project) addresses the unmet need for efficient and reliable cache coherence in multi-core systems. Modern computing is rapidly becoming complex, especially with increasing the number of cores contained within processors, and it becomes increasingly challenging to ensure consistency in data across multiple caches used by these cores.

In multicore processors, in which typically every core keeps a private cache that stores frequently accessed data to speed up the processing, the problem of cache coherence arises when more than one core share the same memory space [1]. That is to say, it ensures that all cores have a coherent and consistent view of memory.

In the absence of a good coherence protocol, unpredictability in the behavior of the system could arise. For example: **Inconsistencies**: One core may read stale or incorrect values from shared memory if another core updates shared memory but doesn't invalidate other caches.

**Performance degradation**: Problems in the design of the cache coherence protocols can lead to much waiting time to receive memory updates that cause stalls, loss of efficiency, and waste of computational resources. Deadlocks and Livelocks: Poorly designed protocols can lead to deadlocks, in which further progress is impossible; or livelocks, where systems end up being stuck in an infinite loop. These issues seem to intensify as the systems scale up- processors are moving from quad core to tens or hundreds of cores as are found in current processors [6].

The problem of cache coherence is of extreme relevance, especially keeping in mind the rising significance of multi-core processors in high-performance computing, mobile devices, data centers, and cloud infrastructure. A few relevant statistics underline the importance of this problem:

**Core Count Growth:** A processor type that used to be limited to a single core ten years ago is now a multi-core design. This has caused modern desktop processors such as Intel and AMD to reach 16 cores in their best consumer microprocessors whereas server processors, in the case of AMD's EPYC 9004, can support up to 128 cores[7]. These counts in more frequently occurring and complex cache coherence problems.

Our specific problem for research addresses these hardware-level implementation difficulties:

- RTL-based multi-core L1 cache design that deploys the MESI protocol.
- Race condition, transient states, and non-atomic state transitions correctness issues.
- Simulation Based techniques to ensure that hardware implementation is correct and deadlock-free.
- Compare verification results with traditional UVM-based simulation techniques.

The demand for efficient cache coherence crosses multiple industries, from Intel and AMD - the world's two leading processor manufacturers - to a multibillion-dollar market for increasingly complex multi-core systems. Every data center and cloud provider, from Amazon AWS and Google Cloud to the \$350 billion sector[9], needs effective coherence for better performance and consumption at reduced power levels. Therefore, the coherence for mobile and embedded systems by Qualcomm and Apple depends on maintaining performance in a \$300 billion market. HPC and AI research further depend on coherence for parallel workloads, which has fostered growth in a \$50 billion industry.

# Chapter 3

# Proposed Approach and Objective

The project offers the implementation of a quad-core system that uses the MESI protocol to tackle the problem of cache coherence. Our implementation maintains all forms of correct data synchronization across multiple cores, each equipped with private L1 caches and a shared L2 cache. Communication occurs through an atomic snooping bus, an arbiter to control bus access, and a directly-mapped algorithm for cache replacement. The primary focus is on L1 cache operations in coordination with the L2 cache. In this context, the methodology aims to be both efficient and reliable.

## 3.1 Proposed Solution

The design is centered on a snooping-based MESI protocol for managing cache coherence. Each core in the system includes a private L1 cache, while a shared L2 cache handles interactions with main memory. The atomic nature of the snooping bus ensures that only one transaction can occur at any given time, simplifying the enforcement of coherence and enhancing overall system performance [6].

## 3.2 Key Value Proposition

- **Configurability:** Our design allows for parameterized core and cache configurations, making it adaptable to various system requirements.
- **Robust Coherence Protocol:** The MESI protocol ensures reliable coherence management, preventing inconsistencies between cores and preserving data integrity.
- **Balanced Performance:** While correctness and coherence are critical, our design also emphasizes balanced performance through the use of arbitration and write-back strategies [8, 9].

## 3.3 Project Objectives

- Implement a quad-core system featuring private L1 caches and a shared L2 cache.
- Design an atomic snooping bus for efficient communication and coherence enforcement.
- Apply the MESI protocol to maintain data consistency across all cores.
- Provide a parameterized, reusable design that supports future scalability and enhancements.
- Validate and verify the design using a UVM-based verification methodology [11].

Additionally, the RISC-V Sodor framework provides a valuable foundation for implementing our quad-core architecture, demonstrating adaptability across a range of system requirements [10].

## 3.4 Flow Diagram

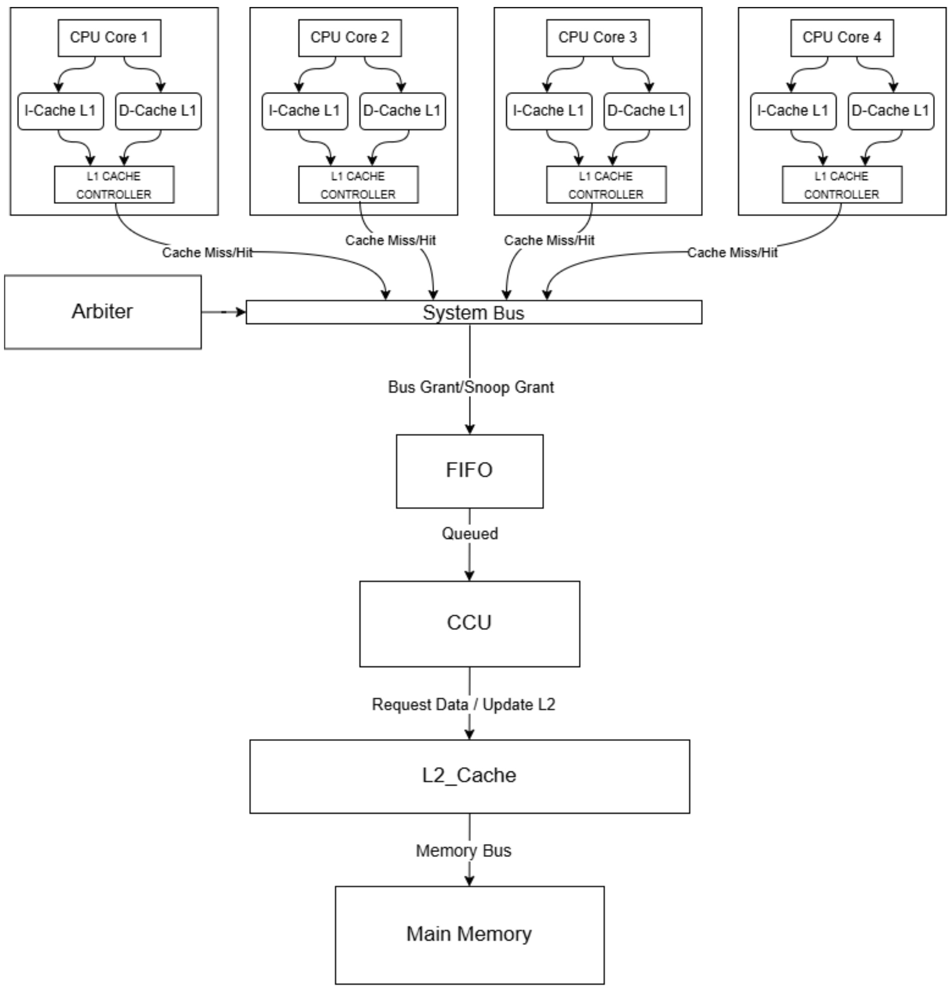


**Figure 3.1:** Block Diagram of the Quad-Core System Architecture

The figure represents the data flow and coherence management in a quad-core processor system implementing the MESI protocol. Each CPU core is equipped with private L1 instruction and data caches, controlled by an L1 cache controller. The sequence of operations and interactions is as follows:

1. **Core Access:** Each CPU core (Core 1 to Core 4) performs memory accesses through its L1 I-Cache and D-Cache, managed by the L1 Cache Controller.
2. **System Bus Communication:** If a cache miss (or coherence action) occurs, the core sends a request to the **System Bus**. The request includes read/write operations or snoop messages.
3. **Arbitration:** The **Arbiter** manages access to the shared System Bus, ensuring that only one core can transmit on the bus at a time. It issues a **Bus Grant/Snoop Grant** based on priority or fairness algorithms.
4. **FIFO Queuing:** Granted transactions are enqueued into a **FIFO** buffer for ordering and temporary storage before being processed.
5. **Cache Control Unit (CCU):** The CCU manages the coherence protocol, particularly the MESI protocol. It processes the queued requests, performs snoop operations if required, and coordinates cache line ownership.
6. **L2 Cache Interaction:** The CCU communicates with the **L2 Cache** to fetch data on cache misses or update it on write-back operations. The L2 Cache is shared among all cores and acts as the last-level cache before main memory access.
7. **Main Memory Access:** If the required data is not present in the L2 Cache, the CCU forwards the request through the **Memory Bus** to access **Main Memory**.

This hierarchical structure ensures efficient and coherent memory access across multiple cores, leveraging the MESI protocol, a centralized arbiter, FIFO-based queuing, and a shared L2 cache for scalability and correctness.

## 3.5 Block Diagram

A quad-core processor system with a memory hierarchy is depicted in Figure 3.2. This is how the data flow operates:

1. A core uses the **L1 Cache Controller** to first check the L1 cache when it requires data.
2. If the L1 cache is missed, the request is routed through the **Cache Coherence Unit (CCU)** to the L2 cache.
3. The request moves to **Main Memory** in the event of an L2 cache miss.

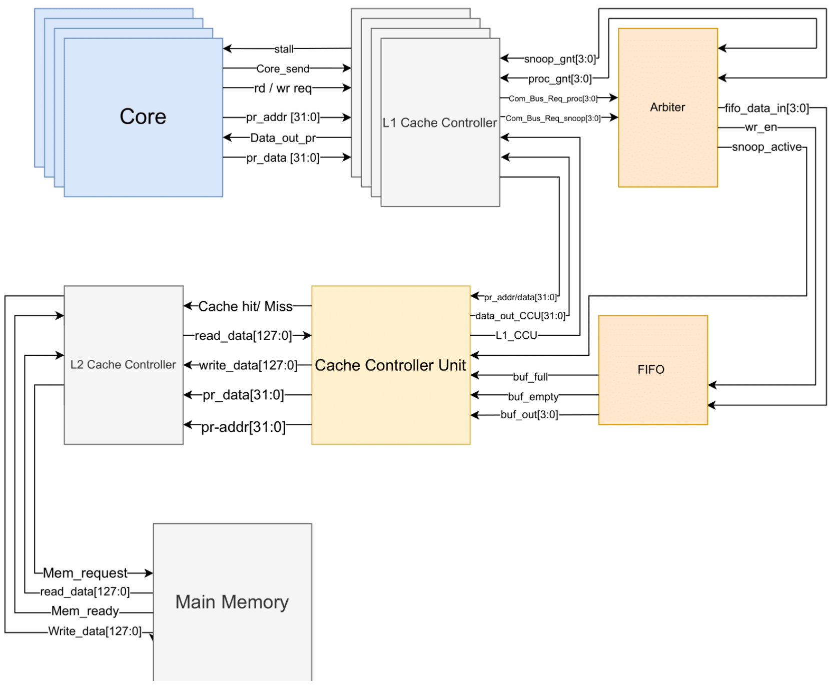


**Figure 3.2:** Block Diagram

4. When several cores request resources, the **Arbiter** makes sure that access is ordered.

Control signals that help maintain cache coherence between cores include `Bs Grant` and `Snoop Grant`. With near caches (L1) being faster but smaller and farthest memories (L2, Main Memory) being slower but larger, this hierarchical design aids in balancing processing performance and memory access latency.

# Chapter 4

# Project Implementation

The implementation of a quad-core system that uses the MESI protocol maintains all kinds of correct data synchronizations across multiple cores that have private L1 caches and a shared L2 cache. The communication is through the atomic snooping bus, an arbiter for the control of bus access, and a directly mapped algorithm for the replacement in caches. We will focus all our attention on the L1 cache by contacting L2. In this scenario, the methodology should be efficient and reliable.

## 4.1 Core Components

**1.Core**

Each core in the system generates read and write requests that are directed to the L1 cache. Both the address `pr_addr` and data `pr_data` fields of the request are used as a message sent to the L1 cache controller. It also gets stalled if L1 Cache Controller is currently working on previous request.
**Deliverables:** It is use to handle generation of memory requests in the processor. Communicate with the L1 cache controller to maintain coherence based on the MESI protocol [2].

**2. L1 Cache Controller**
The L1 cache controller manages the data in the L1 cache for each core. Processes read and write requests sent by the core, verifies whether a read or write request is a cache hit or miss, and takes care of communication with the arbiter.

**Deliverables:** Cache hits and misses. Coordinates with the arbiter to have bus access.

**3. Arbiter**

The arbiter monitors the access to the system bus; it permits only one core or cache controller to use the bus at a time. Arbiter grants are based on requests from the cores and the L1 cache controller.

**Deliverables:** Guarantee priority wise grant access to the system bus. Suppress race conditions; permit bus access to one core at a time upon priority basis.

**4. FIFO (First In, First Out)**

The FIFO buffer will hold the waiting requests from the cores or cache controllers for some time if the bus is busy or waiting to be accessed. It works with the arbiter to handle requests efficiently.

**Deliverables:** Queue pending memory requests. Manage request ordering and forwarding to the CCU.

**5. Cache Controller Unit**

The cache controller unit acts to coordinate L1 cache misses and manage interaction with the shared L2 cache. It resolves cache misses, takes care of write-backs, and retrieves proper data from memory. It decodes the `buf_out` signal and identify which core requested and then process to entertain that request.

**Deliverables:** Manage the interaction between L1 and L2. Enforce cache coherence by snooping bus transactions and enforcing MESI states.

**6. L2 Cache Controller**

The L2 cache controller acts as a bridge between the L1 cache and the main memory. It holds data as well as instructions, and with the support of the cache controller unit, memory requests are handled.

**Deliverables:** Receive and manage shared data among all cores. Cache miss handling, write-backs, and data forwarding to L1 caches.

**7. Main Memory**

The last level is the main memory where the data is sent in case of a hit or miss in L1 as well as in L2 caches. Here, the read and write requests to the main memory are carried out from the controller of the L2 cache.

**Deliverables:** Process memory requests in situations where access from the L2 cache cannot be served as a hit. Maintain memory consistency with the hierarchy of the cache. Its efficient operation is critical for overall system performance, as it ensures that all core requests are fulfilled without unnecessary delays [5].

| **Parameter** | **Description** | **Value** |
|---|---|---|
| Number of Cores | Total number of CPU cores | 4 (quad-core) |
| L1 Cache Size (per core) | Private cache size for each core | 4 KB |
| L2 Cache Size | Shared cache size across all cores | 8 KB |
| Coherence Protocol | Cache coherence protocol used | MESI |
| Bus Type | Communication bus between cores | Snooping,priority |

**Table 4.1:** Parameter and Configuration

## 4.2 State Diagram

### 4.2.1 L1 Cache Controller

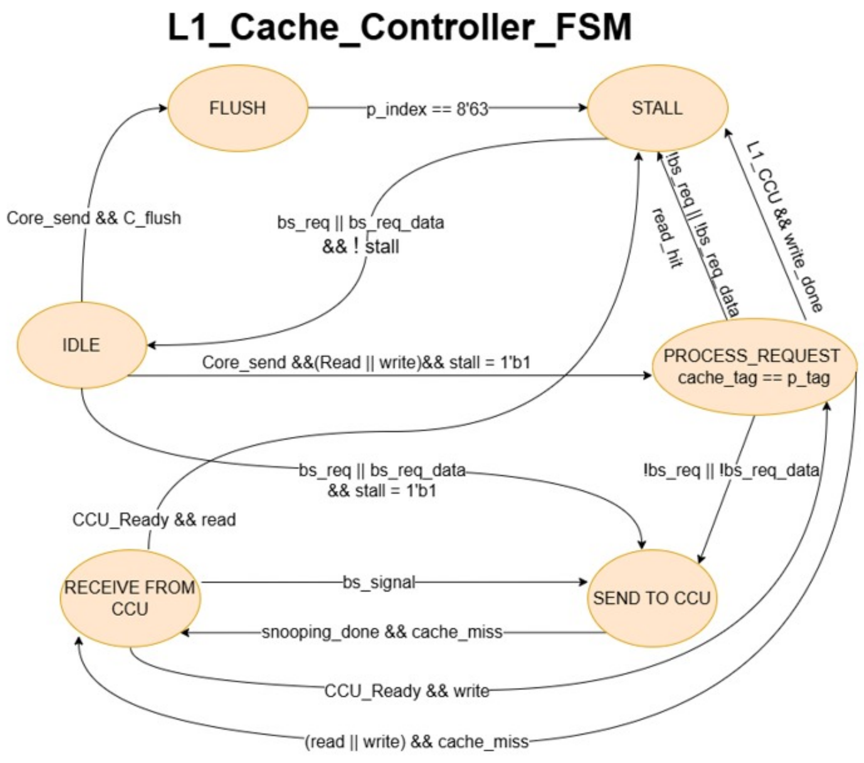


**Figure 4.1:** Finite State Machine of the L1 Cache Controller

The above state diagram represents the finite state machine (FSM) of the L1 Cache Controller. It controls how requests from the core (read, write, or flush) are processed based on the current state and coherence-related signals. The FSM starts in the `IDLE` state, evaluates core requests, and transitions accordingly through states such as `PROCESS_REQUEST`, `SEND TO CCU`, `RECEIVE FROM CCU`, and `FLUSH`. The `STALL` state ensures that no conflicting operations occur when the system is not ready. Coherence events such as snooping or shared bus data (e.g., `bs_req`, `bs_signal`) influence transitions, particularly in cache miss scenarios. This design ensures orderly and synchronized cache behavior in a multicore environment using the MESI protocol.

### 4.2.2 L2 Cache Controller

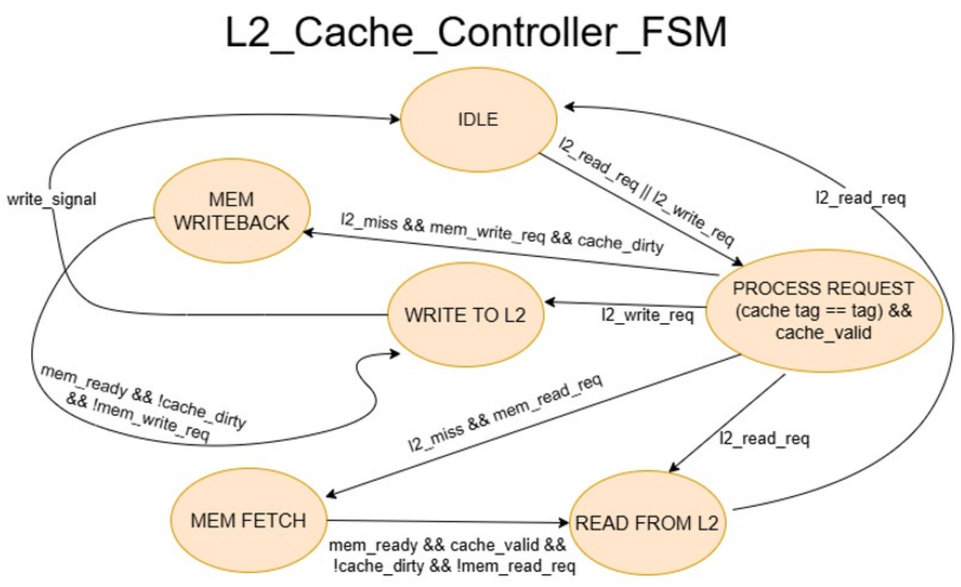

L2_Cache_Controller_FSM
IDLE
MEM WRITEBACK
WRITE TO L2
PROCESS REQUEST (cache tag == tag) && cache_valid
MEM FETCH
READ FROM L2
write_signal
l2_read_req || l2_write_req
l2_read_req
l2_miss && mem_write_req && cache_dirty
l2_write_req
mem_ready && !cache_dirty && !mem_write_req
l2_miss && mem_read_req
l2_read_req
mem_ready && cache_valid && !cache_dirty && !mem_read_req

**Figure 4.2:** Finite State Machine of the L2 Cache Controller

The FSM of the L2 Cache Controller coordinates memory access operations based on cache hits, misses, and the validity of data. It starts in the `IDLE` state and transitions to `PROCESS REQUEST` upon receiving read or write commands. If a write is valid and hits, it proceeds to `WRITE TO L2`; otherwise, in case of a miss and dirty line, it transitions to `MEM WRITEBACK` before writing. If a memory read is needed, it transitions to `MEM FETCH`, and on successful fetch with valid data, it reads from L2 (`READ FROM L2`). This ensures data consistency between L2 and main memory while handling simultaneous read/write requests efficiently.

### 4.2.3 Cache Controlling Unit (CCU)

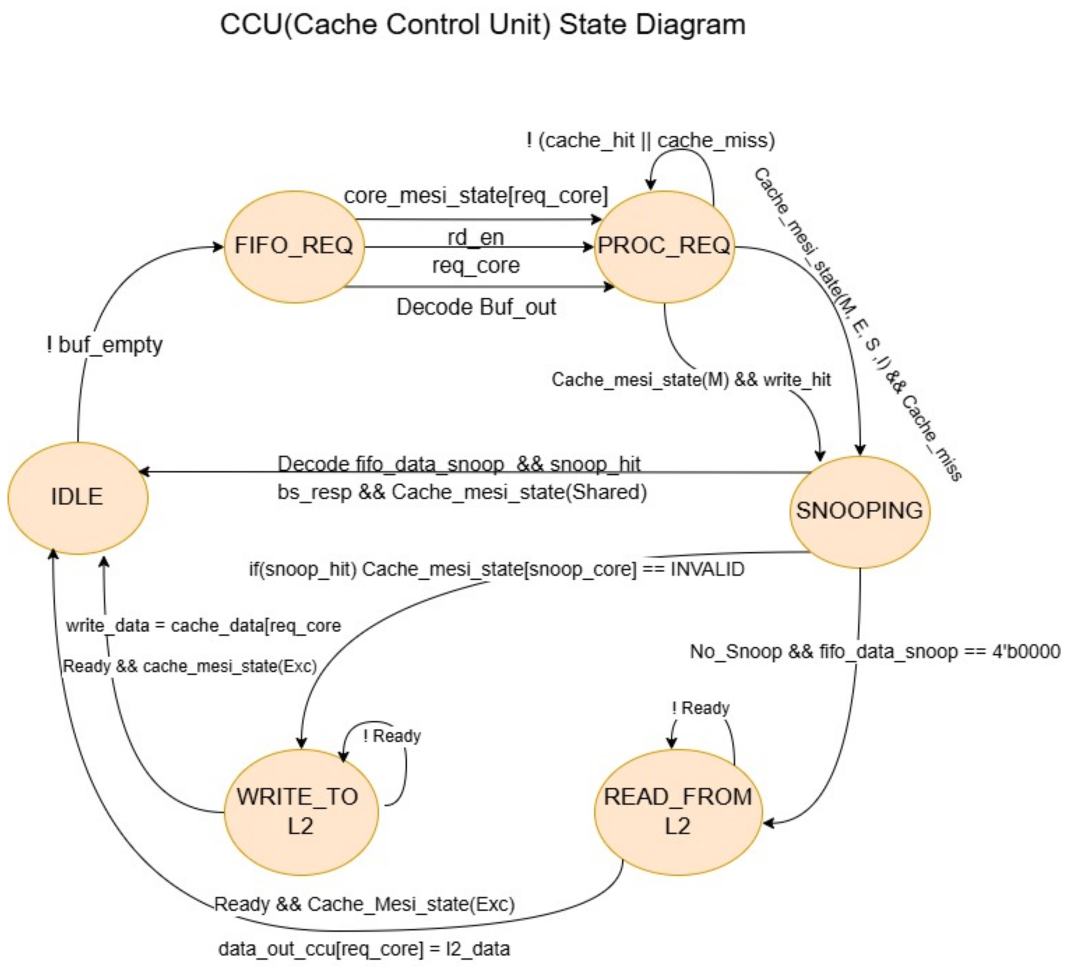

CCU(Cache Control Unit) State Diagram
! (cache_hit || cache_miss)
core_mesi_state[req_core]
rd_en
req_core
Decode Buf_out
FIFO_REQ
PROC_REQ
Cache_mesi_state(M, E, S, I) && Cache_miss
! buf_empty
Cache_mesi_state(M) && write_hit
IDLE
Decode fifo_data_snoop && snoop_hit
bs_resp && Cache_mesi_state(Shared)
SNOOPING
if(snoop_hit) Cache_mesi_state[snoop_core] == INVALID
write_data = cache_data[req_core
Ready && cache_mesi_state(Exc)
No_Snoop && fifo_data_snoop == 4'b0000
! Ready
! Ready
WRITE_TO L2
READ_FROM L2
Ready && Cache_Mesi_state(Exc)
data_out_ccu[req_core] = l2_data

**Figure 4.3:** Finite State Machine of the Cache Coherence Unit (CCU)

The Cache Control Unit (CCU) finite state machine (FSM) manages cache coherence using the MESI protocol. It transitions between several states based on cache hits/misses, snoop hits, and MESI states of the requesting and snooping cores. The key states are:

- `IDLE`: Waits for new requests from the processor or the snoop network.
- `FIFO_REQ`: Receives a queued request from the FIFO and prepares for processing.
- `PROC_REQ`: Processes the request by checking the cache tag and MESI state.
- `SNOOPING`: Engages in snoop operations to validate or invalidate lines in other caches.
- `READ_FROM_L2`: Fetches a cache line from L2 memory in the case of a miss or an invalid MESI state.
- `WRITE_TO_L2`: Writes back modified data to L2 if required by the coherence protocol.

The typical flow begins in the `IDLE` state, awaiting new requests. When a request is detected, the FSM transitions to `FIFO_REQ`, then to `PROC_REQ` for processing. If the request results in a cache miss or MESI validation is needed, the FSM may enter the `SNOOPING` state. Depending on the snoop results, it may transition to either `READ_FROM_L2` or `WRITE_TO_L2`. Once the data operation is completed, the FSM returns to the `IDLE` state, ensuring coherence is maintained across cores.

# Chapter 5

# Testing and Verification

## 5.1 UVM Based Verification

**Overview and Significanc** The **Universal Verification Methodology (UVM)** is a standardized framework used for verifying complex digital designs, including multi-core cache coherence systems. UVM provides a structured, reusable, and scalable environment for functional verification. In our quad-core system implementing the **MESI protocol**, UVM plays a critical role in validating cache coherence, bus arbitration, memory consistency, and protocol compliance.

**Benefits of Using UVM for Verification**

- **Scalable and Reusable:** Enables modular testbench components, making it easy to extend for different configurations.
- **Randomized and Functional Coverage:** Ensures exhaustive testing of various corner cases and system-level behaviors.
- **Automation and Debugging:** Built-in features like Transaction-Level Modeling (TLM) and a message reporting system simplify debugging [6].

**Resources in UVM**

- **Base Classes:** UVM provides pre-defined classes such as `uvm_component`, `uvm_test`, `uvm_scoreboard`, and `uvm_monitor`, forming the foundation of the verification environment.
- **Transaction-Level Modeling (TLM):** Enables communication between components through ports, exports, and analysis ports, allowing flexible data exchange.
- **Sequence and Sequencer:** Manage stimulus generation, enabling the creation and reuse of complex test scenarios.

- **Configuration Mechanism:** Uses `uvm_config_db` to allow dynamic configuration of the verification environment, making it adaptable for different test cases.
- **Coverage and Scoreboarding:** Provides built-in support for functional coverage and scoreboarding to track verification progress and ensure data integrity.
- **Report and Logging:** Includes a reporting system for logging events, errors, and debug information to aid analysis and debugging.
- **Reference Implementation:** Maintained by Accellera, the UVM library includes examples and documentation to assist users.

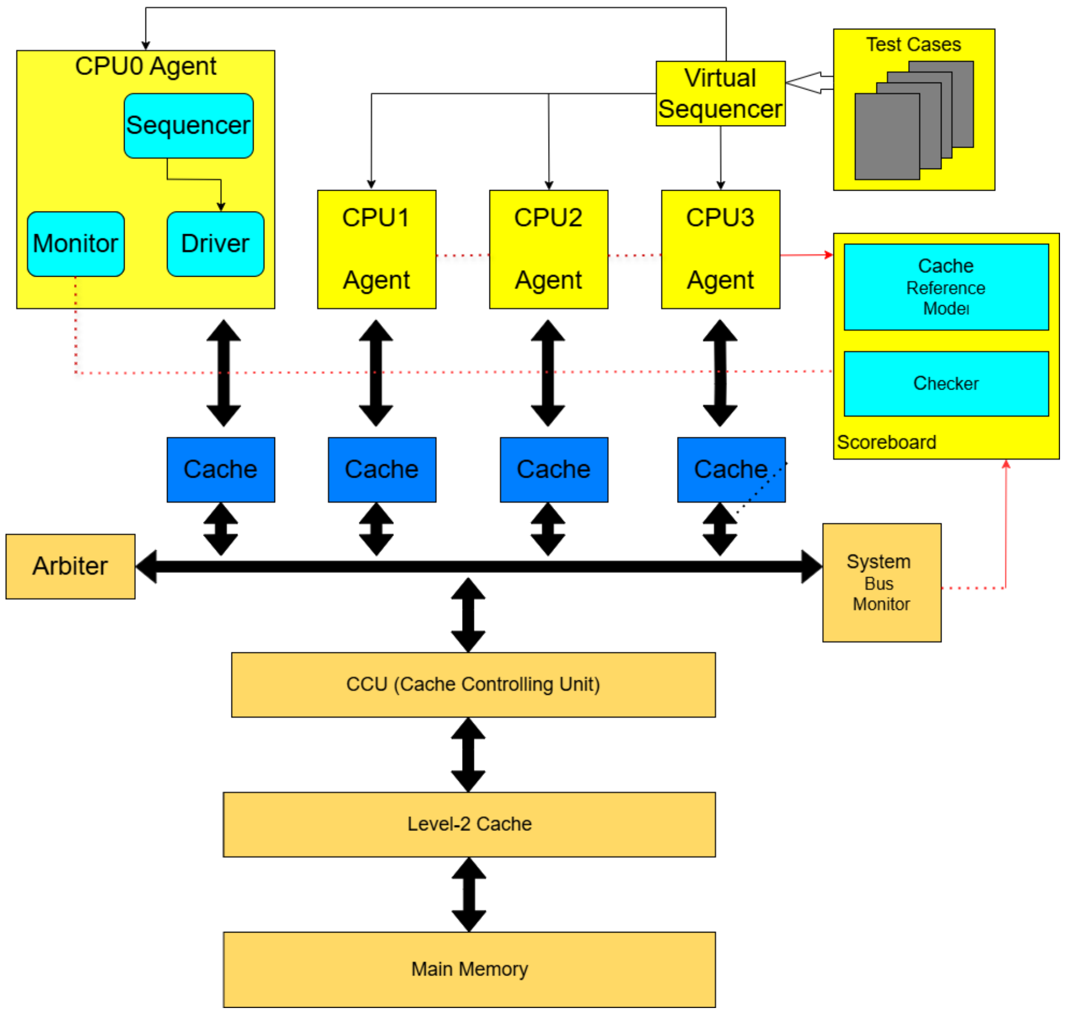


**Figure 5.1:** Block Diagram of UVM

**Quad-Core Cache Coherence Verification Block Diagram** The block diagram illustrates a verification setup for a quad-core processor with cache coherence, ensuring all CPU caches (CPU0, CPU1, CPU2, CPU3) maintain a consistent view of data.

**Component Breakdown**

- **CPU0 Agent Sequencer:** Generates operation sequences (e.g., reads or writes) to simulate CPU behavior, connected to a driver that sends these operations to the CPU.
- **Monitor and Driver:** The monitor observes activity between the sequencer and CPUs, while the driver translates sequences into actual CPU signals.
- **CPU1, CPU2, CPU3 with Agents:** Each CPU has an agent acting as a smart controller for managing interactions and local cache access.
- **Caches:** Each CPU has its own L1 cache (represented by blue boxes), storing copies of data from main memory for faster access.
- **Arbiter:** Acts as a traffic controller, granting access to the system bus based on priority or fairness, preventing conflicts between CPU requests.
- **Cache Controlling Unit (CCU) and Level-2 Cache:** The CCU manages coherence and handles data consistency operations. The L2 cache serves as a shared cache between CPU caches and main memory.
- **Main Memory:** The final level of storage, accessed by the L2 cache and CCU when data is not found in the lower levels.
- **Virtual Sequencer and Test Cases:** The virtual sequencer coordinates multi-core test scenarios using a library of test cases to verify cache coherence.
- **Cache Reference Model and Checker:** Simulates ideal cache behavior. The checker compares actual results against the model to detect errors.
- **Scoreboard:** Tracks and compares transactions across components, verifying data consistency throughout the system.
- **System Bus Monitor:** Observes transactions on the system bus to ensure correct and efficient data flow.
- **System Monitor:** Oversees the entire verification environment, logging performance and detecting system-level issues.

This verification environment ensures that when one CPU updates its cache, all other CPUs are notified and update their caches accordingly. This mechanism enforces cache coherence across the quad-core system.

## 5.2 Results and Analysis

This section presents the experimental findings based on various test cases designed to evaluate the performance and efficiency of the Quad-core architecture using MESI protocol for cache consistency. The primary focus of the analysis is to evaluate the efficiency of implemented protocol in maintaining data consistency. Through different case checks, systematic simulation and verification, the effectiveness of design was evaluated.

### 5.2.1 Read Test Cases

| **Test Scenario** | **Core Address** | **Arbitration Order** | **State Transition** | **Verification Method** |
|---|---|---|---|---|
| **All Cores Read** | 32'h11111000 | Core 0 → Core 1<br>Core 2 → Core 3 | • Core 0: Miss → Exclusive → Shared<br>• Cores 1–3: Miss → Shared (via snooping) | • Basic Testing in Vivado<br>• UVM Scoreboard |
| **Split Address Read** | Core 0,1: 32'h11111000<br>Core 2,3: 32'h11110000 | Core 0 → Core 1<br>Core 2 → Core 3 | • Core 0/2: Miss → Exclusive→ Shared<br>• Cores 1/3: Miss → Shared (via snooping) | • Basic Testing in Vivado<br>• UVM Scoreboard |
| **Random Read** | Random Addr | Priority-based | Data read based on offset matching (via snooping) | UVM Scoreboard |

**Table 5.1:** Summary of Read Test Cases for Cache Coherence System

**Example Waveform Analysis** **Case 1:** If all cores have the same address `32'h11111000` and `read` is asserted.

When Core 0 encounters a compulsory cache miss, it generates a request sent to the arbiter, which grants access based on priority. The Cache Coherence Unit (CCU) reads requests from the FIFO, communicates with the L1 cache controller, and initiates snooping. Since all cores experience a read miss at the same address, an L2 read request is made to fetch data. Core 0 receives the data in the **EXCLUSIVE** state, then shares it with other cores, transitioning them to the **SHARED** state. This ensures efficient data consistency across the quad-core system following the MESI protocol.

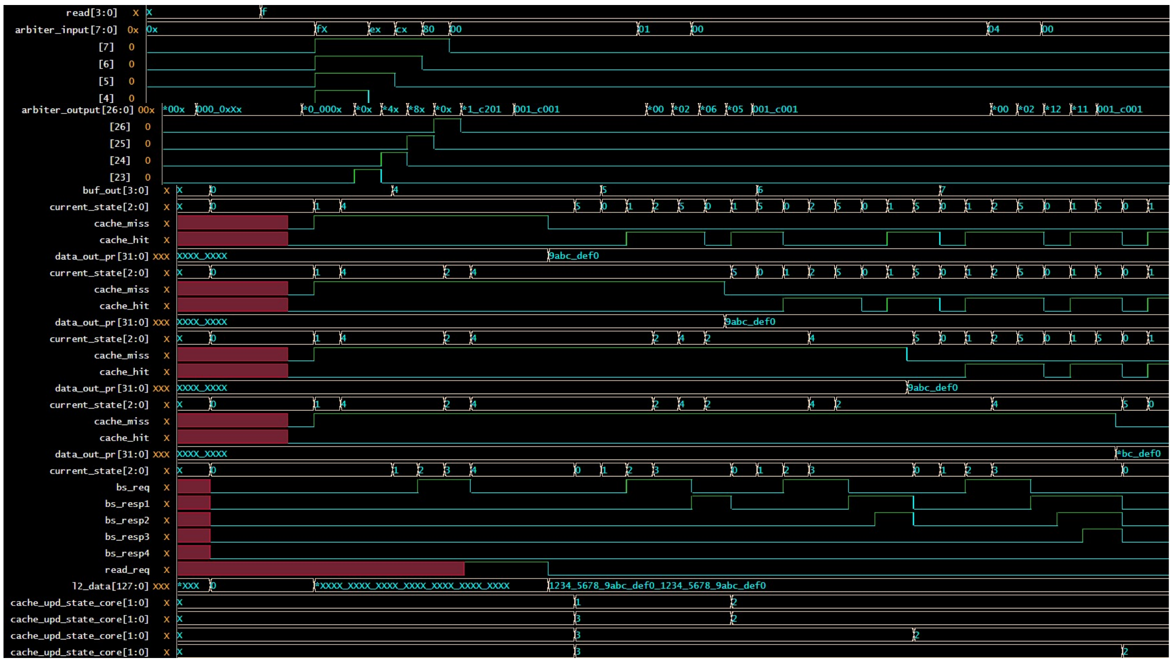


**Figure 5.2:** Waveform of all cores reading from the same address

### 5.2.2 Write Test Cases

| **Test Scenario** | **Core Address** | **Arbitration Order** | **State Transition** | **Verification Method** |
|---|---|---|---|---|
| **All Cores Write** | 32'h11111000 | Core 0 → Core 1<br>Core 2 → Core 3 | • Initial: Miss → Exclusive<br>• Then: Write Hit → Modified<br>• Other Cores: Invalidate | • Basic Testing in Vivado<br>• UVM Scoreboard |
| **Mixed Write/Read** | • Cores 0,1: 32'h11111000 (Write)<br>• Cores 2,3: 32'h11110000 (Read) | Core 0 → Core 1<br>Core 2 → Core 3 | • Write: Miss → Hit → Modified<br>• Read: Miss → Shared (via snooping)<br>• Invalidate other caches as needed | • Basic Testing in Vivado<br>• UVM Scoreboard |
| **Random Write** | Random Addr | Priority-based | • State transitions follow MESI (Write) | UVM Scoreboard |

**Table 5.2:** Summary of Write Test Cases for Cache Coherence System

**Example Case 3: Random Write Across Cores with Random Address**

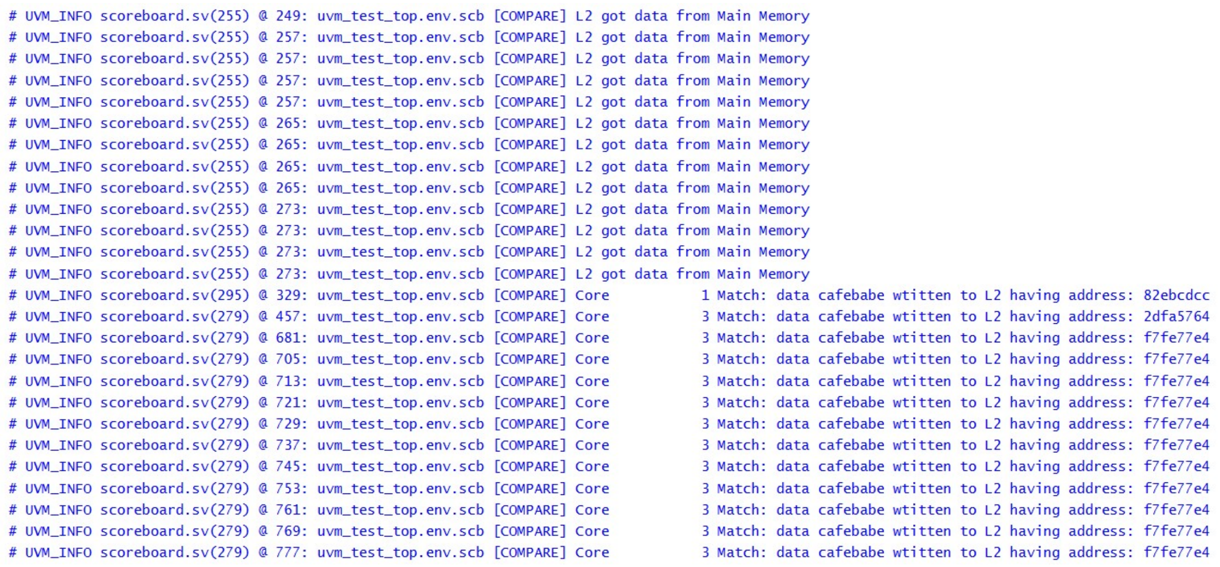
In this case, random write requests are generated across multiple cores with random memory addresses. To ensure data integrity, we implemented a UVM scoreboard check that verifies whether data is correctly written to the L2 cache.

The following procedure was followed:

- The `pr_data` signal was initialized to `32'hCAFEBABE`.
- A comparison was made between the written data and the expected data using the offset.
- If all write cases pass, this verifies that the data is written correctly to the L2 cache.

The verification ensures that random write requests are handled properly across all cores and that data is correctly stored in the cache.

The UVM log file of performing cheks of scoreboard is hown below:



```
# UVM_INFO scoreboard.sv(255) @ 249: uvm_test_top.env.scb [COMPARE] L2 got data from Main Memory
# UVM_INFO scoreboard.sv(255) @ 257: uvm_test_top.env.scb [COMPARE] L2 got data from Main Memory
# UVM_INFO scoreboard.sv(255) @ 257: uvm_test_top.env.scb [COMPARE] L2 got data from Main Memory
# UVM_INFO scoreboard.sv(255) @ 257: uvm_test_top.env.scb [COMPARE] L2 got data from Main Memory
# UVM_INFO scoreboard.sv(255) @ 257: uvm_test_top.env.scb [COMPARE] L2 got data from Main Memory
# UVM_INFO scoreboard.sv(255) @ 265: uvm_test_top.env.scb [COMPARE] L2 got data from Main Memory
# UVM_INFO scoreboard.sv(255) @ 265: uvm_test_top.env.scb [COMPARE] L2 got data from Main Memory
# UVM_INFO scoreboard.sv(255) @ 265: uvm_test_top.env.scb [COMPARE] L2 got data from Main Memory
# UVM_INFO scoreboard.sv(255) @ 265: uvm_test_top.env.scb [COMPARE] L2 got data from Main Memory
# UVM_INFO scoreboard.sv(255) @ 273: uvm_test_top.env.scb [COMPARE] L2 got data from Main Memory
# UVM_INFO scoreboard.sv(255) @ 273: uvm_test_top.env.scb [COMPARE] L2 got data from Main Memory
# UVM_INFO scoreboard.sv(255) @ 273: uvm_test_top.env.scb [COMPARE] L2 got data from Main Memory
# UVM_INFO scoreboard.sv(255) @ 273: uvm_test_top.env.scb [COMPARE] L2 got data from Main Memory
# UVM_INFO scoreboard.sv(295) @ 329: uvm_test_top.env.scb [COMPARE] Core          1 Match: data cafebabe wtitten to L2 having address: 82ebcdcc
# UVM_INFO scoreboard.sv(279) @ 457: uvm_test_top.env.scb [COMPARE] Core          3 Match: data cafebabe wtitten to L2 having address: 2dfa5764
# UVM_INFO scoreboard.sv(279) @ 681: uvm_test_top.env.scb [COMPARE] Core          3 Match: data cafebabe wtitten to L2 having address: f7fe77e4
# UVM_INFO scoreboard.sv(279) @ 705: uvm_test_top.env.scb [COMPARE] Core          3 Match: data cafebabe wtitten to L2 having address: f7fe77e4
# UVM_INFO scoreboard.sv(279) @ 713: uvm_test_top.env.scb [COMPARE] Core          3 Match: data cafebabe wtitten to L2 having address: f7fe77e4
# UVM_INFO scoreboard.sv(279) @ 721: uvm_test_top.env.scb [COMPARE] Core          3 Match: data cafebabe wtitten to L2 having address: f7fe77e4
# UVM_INFO scoreboard.sv(279) @ 729: uvm_test_top.env.scb [COMPARE] Core          3 Match: data cafebabe wtitten to L2 having address: f7fe77e4
# UVM_INFO scoreboard.sv(279) @ 737: uvm_test_top.env.scb [COMPARE] Core          3 Match: data cafebabe wtitten to L2 having address: f7fe77e4
# UVM_INFO scoreboard.sv(279) @ 745: uvm_test_top.env.scb [COMPARE] Core          3 Match: data cafebabe wtitten to L2 having address: f7fe77e4
# UVM_INFO scoreboard.sv(279) @ 753: uvm_test_top.env.scb [COMPARE] Core          3 Match: data cafebabe wtitten to L2 having address: f7fe77e4
# UVM_INFO scoreboard.sv(279) @ 761: uvm_test_top.env.scb [COMPARE] Core          3 Match: data cafebabe wtitten to L2 having address: f7fe77e4
# UVM_INFO scoreboard.sv(279) @ 769: uvm_test_top.env.scb [COMPARE] Core          3 Match: data cafebabe wtitten to L2 having address: f7fe77e4
# UVM_INFO scoreboard.sv(279) @ 777: uvm_test_top.env.scb [COMPARE] Core          3 Match: data cafebabe wtitten to L2 having address: f7fe77e4
```

**Figure 5.3:** UVM Log File

The log shown is part of a UVM-based (Universal Verification Methodology) simulation for a cache coherence testbench. It highlights interactions between the Level 2 (L2) cache and main memory, as well as verification comparisons performed in a scoreboard component. Each transaction is monitored to ensure that data received from L2 reflects the most recent and valid memory state, especially during cache miss scenarios. When a read or write occurs, the fetched data is captured and checked against expected values stored in the scoreboard to verify consistency. This ensures that the cache coherence protocol maintains data accuracy across all cores, even under simultaneous access conditions.

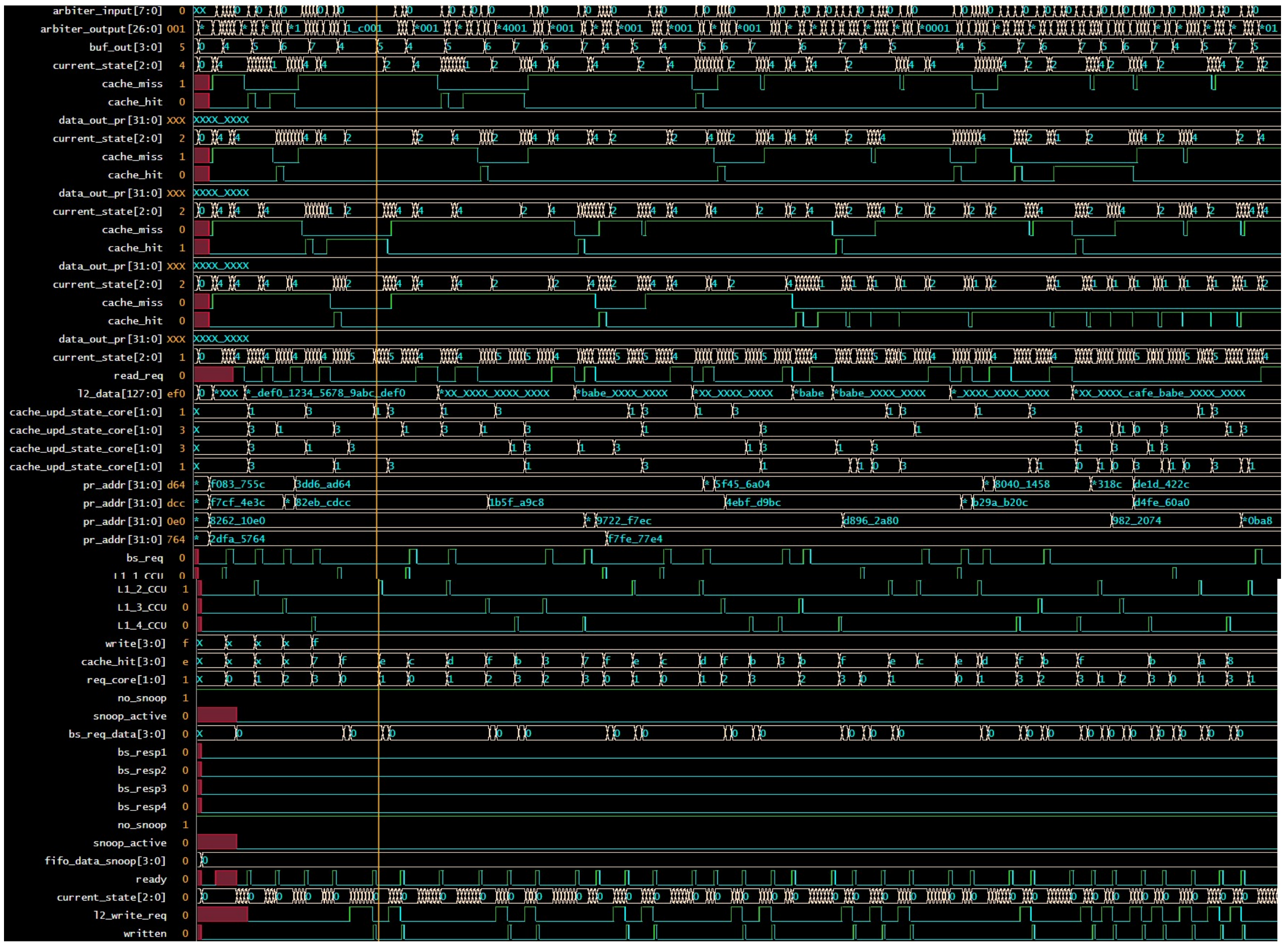


**Figure 5.4:** Waveform of Random write

### 5.2.3 Corner Case in UVM Verification

**Case: Concurrent Read-Write Scenario with Shared Addresses** This scenario tests the coherence mechanism under concurrent read and write operations across multiple cores accessing shared addresses.

- Core 0 and Core 1 access the same memory address, while Core 2 and Core 3 access a different but shared address.
- When a write is asserted by Core 0, it encounters a compulsory miss and generates a request.
- This request is queued in the FIFO after arbitration.
- The Cache Coherence Unit (CCU) processes Core 0's request first, snooping for data. As no matching data is found, it fetches the block from L2 and shares it with Core 1.
- Core 2 follows a similar flow: after a miss, the CCU fetches the block from L2 and shares it with Core 3.

- Later, Core 0 gets a write hit on the same address and generates a write request.
- After passing through arbitration, the CCU invalidates Core 1's cache line before writing the updated data to L2 in a write-through fashion.

This case demonstrates the effectiveness of the MESI protocol and the system's ability to maintain data coherence during simultaneous accesses to shared memory addresses.

| Cores | Pr_Addr[31:0] | Write/Read | Hit/miss | State | Hit/miss | State | Hit/miss | State |
|---|---|---|---|---|---|---|---|---|
| Core0 | 32'h11111000 | Write | Miss | E, S, I | hit | S, M, E | Miss | E,I |
| Core1 | 32'h11111000 | Write | Miss | I, S, I | miss | S, I | Hit | I,M,E |
| Core2 | 32'h11110000 | Read | Miss | I, E, S | hit | S | Hit | S |
| Core3 | 32'h11110000 | Read | Miss | I, I, S | hit | S | Hit | S |

**Table 5.3:** Cache Access Table with Hit/Miss States

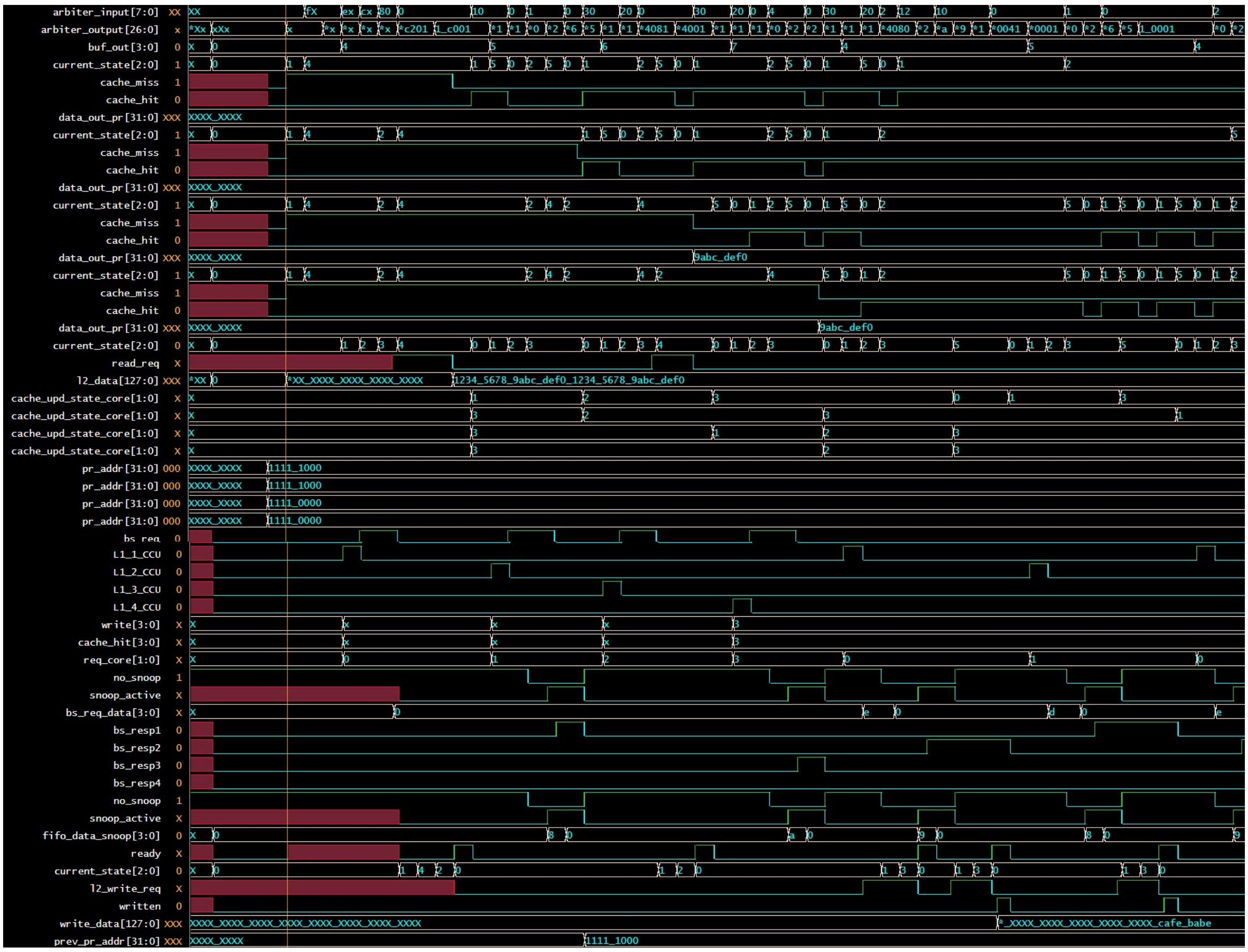
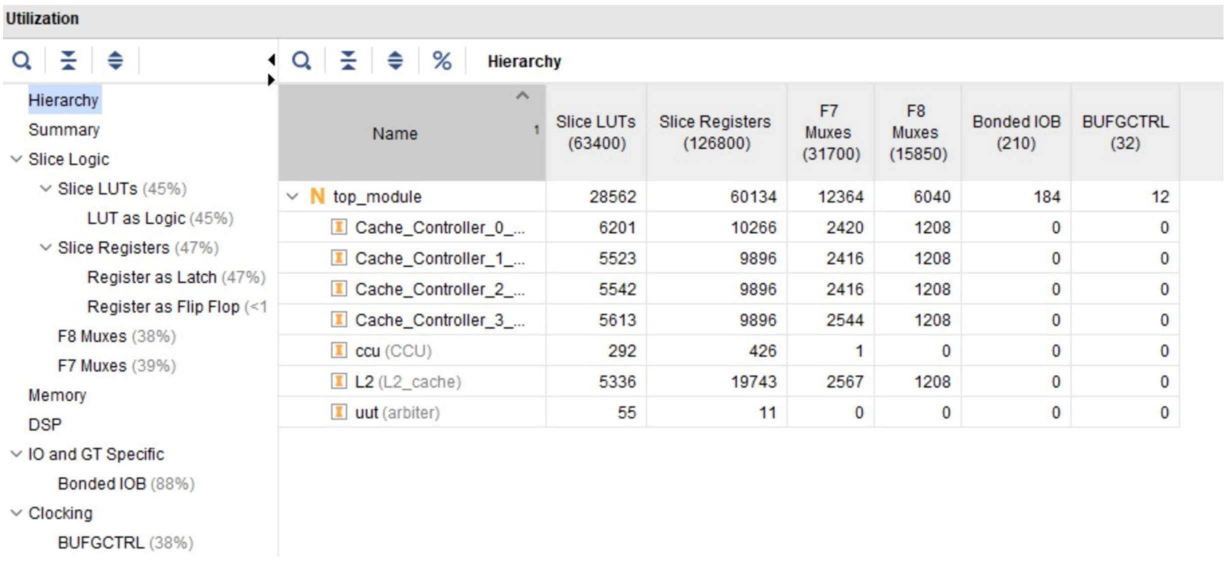


**Figure 5.5:** Waveform of Concurrent Read-Write Scenario with Shared Addresses

## 5.3 Resource Utilization Report for Top-Level Design

The utilization report shows that the `top_module` consumes 28,562 Slice LUTs and 60,134 Slice Registers out of the available FPGA resources. Among submodules, each `Cache_Controller` utilizes approximately 5,500–6,200 LUTs and around 9,800–10,200 registers, indicating their complexity and active role in the design. The `L2_cache` also has significant usage, especially in Slice Registers (19,743), reflecting its centralized memory control. The `CCU` and `arbiter` modules have minimal resource usage, showing their comparatively lightweight logic.



| Name | Slice LUTs (63400) | Slice Registers (126800) | F7 Muxes (31700) | F8 Muxes (15850) | Bonded IOB (210) | BUFGCTRL (32) |
|---|---|---|---|---|---|---|
| top_module | 28562 | 60134 | 12364 | 6040 | 184 | 12 |
| Cache_Controller_0_... | 6201 | 10266 | 2420 | 1208 | 0 | 0 |
| Cache_Controller_1_... | 5523 | 9896 | 2416 | 1208 | 0 | 0 |
| Cache_Controller_2_... | 5542 | 9896 | 2416 | 1208 | 0 | 0 |
| Cache_Controller_3_... | 5613 | 9896 | 2544 | 1208 | 0 | 0 |
| ccu (CCU) | 292 | 426 | 1 | 0 | 0 | 0 |
| L2 (L2_cache) | 5336 | 19743 | 2567 | 1208 | 0 | 0 |
| uut (arbiter) | 55 | 11 | 0 | 0 | 0 | 0 |

**Figure 5.6:** Resource Utilization Report for Top-Level Design

# Chapter 6

# Conclusion and Future Directions

In this project, we effectively created, executed, and confirmed a cache controller that adheres to the MESI (Modified, Exclusive, Shared, Invalid) coherence protocol utilizing SystemVerilog. Our controller was incorporated into a simulated multi-core setting and validated with a UVM-based testbench. The controller managed processor demands, snoop-based coherence signals, and interactions with a central cache coherence unit (CCU). The finite state machine (FSM) developed for the cache controller moved accurately through all MESI states and maintained consistent data behavior in shared memory environments.

The UVM scoreboard logs verify the functional accuracy of the data transactions. For example, regular occurrences of the data pattern `cafebabe` produced by Core 3 and seen in the L2 cache suggest correct functioning of write and snoop management processes. Furthermore, the capability of L2 cache to retrieve data from main memory during misses further confirms the interactions within the memory hierarchy and the management of cache misses.

The project demonstrated significant advantages in modular design methodology and scalable verification infrastructure. The implementation of structured internal data types—including specialized structs for processor requests, snoop inputs, and coherence responses—substantially enhanced code readability and maintainability. Comprehensive simulation scenarios, encompassing random write patterns and complex contention cases, validated the robustness of the architecture under diverse operational conditions.

**Future Directions:** While the current implementation successfully ensures coherence at the L2 level within a quad-core architecture, several promising avenues for enhancement have been identified:

- **Core Scaling:** Extend the design from 4 to 8 cores by refining the arbiter logic and optimizing request handling pathways. This would provide significant performance benefits for parallel computing applications while presenting new challenges in coherence management.

- **Hierarchical Cache Implementation:** Develop a complete L1-L2-L3 cache hierarchy with coherence maintained across all levels, more closely mirroring commercial processor architectures.
- **Processor Integration:** Interface with actual processor cores, potentially leveraging RISC-V or ARM-based platforms, to facilitate hardware-level validation in realistic computational scenarios.
- **Performance Optimizations:** Implement non-blocking caches and explore directory-based coherence mechanisms as alternatives to broadcast-based protocols, potentially reducing system-wide coherence traffic.
- **Enhanced Verification Methodologies:** Supplement simulation-based testing with formal verification techniques to mathematically prove the absence of livelocks or deadlocks in the coherence finite state machine.
- **Snooping Mechanism Refinement:** Optimize the existing snooping protocol to minimize bus traffic through advanced filtering techniques and selective invalidation strategies.

These enhancements would substantially improve both the architectural sophistication and practical applicability of the cache coherence system, positioning it for deployment in more complex computational environments.

# References

[1] J. M. Calandrino and J. H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. *In 2009 21st Euro micro Conference on Real Time Systems*, July 2009.

[2] R. Carlson, S. Kim, and P. N. M. A. K. Coherence and consistency in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 22(7):1000–1010, 2011.

[3] M. Chisholm, N. Kim, B. C. Ward, N. Otterness, J. H. Anderson, and F. D. Smith. Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems. In *In 2016 IEEE Real-Time Systems Symposium (RTSS)*, Nov 2016.

[4] Wan-Rong Gao, Jian-Bin Fang, Chun Huang, Chuan-Fu Xu, and Zheng Wang. wrbench: Comparing cache architectures and coherency protocols on armv8 many-core systems. *JCST*, page 1, 2018.

[5] G. Gracioli and A. A. Frohlich. On the design and evaluation of a real-time operating system for cache-coherent multicore architectures. *Journal of Systems Architecture*, 64:1–15, January 2016.

[6] Mohamed Hassan, Anirudh M Kaushik, and Hiren Patel. Predictable cache coherence for multi-core real-time systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 235–246. IEEE, 2017.

[7] N. Kim, M. Chisholm, N. Otterness, J. H. Anderson, and F. D. Smith. Allowing shared libraries while supporting hardware isolation in multicore real-time systems. In *In 2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2017.

[8] B. Lesage, D. Hardy, and I. Puaut. Shared data caches conflicts reduction for wcet computation in multi-core architectures. In *In 18th International Conference on Real-Time and Network Systems*, Toulouse, France, November 2010.

[9] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *In Ninth European Dependable Computing Conference*, 2012.

[10] The Regents of the University of California. Riscv-sodor. https://www.librecores.org/codelec/riscv-sodor, April 2019.

[11] A. Pyka, M. Rohde, and S. Uhrig. Extended performance analysis of the time predictable on-demand coherent data cache for multi- and many-core systems. In *In 2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, July 2014.