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. Tahir

Department of Electrical Engineering

University of Engineering and Technology Lahore

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

In a multicore system, cache coherency is an essential feature, especially in the context of shared memory model. Although coherency protocols may more generally be proved at the architectural level, some difficulties arise when the cache is implemented in HDL in a multi-processor system. Many modern processors have optimized the communication using the split transaction bus but it has only caused further incrementing of transient states and race conditions which has certainly been a challenge .This is the most important reason why designing as well as verifying cache coherency has become really complicated as well as time taking. These intricacies make cache coherency design and verification a time-consuming activity. Sometimes simulation techniques alone are not sufficient for guaranteeing key properties like memory consistency or the lack of deadlocks, livelock, and starvation. A high level of confidence through simulation alone is tough to attain-in particular when subtle race conditions and failures need identification and resolution. This project will test and validate the functionality of the multi-core level 1 cache design based on snooping MESI protocol using simulation techniques in UVM. By performing verification on the RTL of a multicore system, we look more closely on how the system works and various challenges it encounters. In multicore processors, in which typically every core keeps a private cache, that store frequently accessed data to speed up the processing, the problem of cache coherence arises when more than one core share the same memory space. Various checking scenarios encountered in this project, such as whether these certain types of cache coherence violations exist or bus arbitration fairness and system deadlock, to ensure correctness of the system. This is the case with simulation as well, one form of which is a unit test that may be applied to the separate components (L1 cache, arbiter, and FIFO) but also to the entire system. Using simulation allows us to analyze all possible interactions between cores since this method reveals how the whole system behaves under different conditions.

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.

For example, two cores simultaneously try to write the same value into the same memory location at the same time. Therefore, due care must be taken while implementing the MESI protocol so that it does not fall under the category of race conditions. Transient States: During hardware implementation, a transition from one state to another-on for example, from Shared to Exclusive does not happen in an instant. Instead, it is possible for the system to be in intermediate (transient) states during which it is neither fully in the old state nor the new state. Such transient states have to be managed correctly in such a way that no core reads or writes stale or invalid data.Split-Transaction Buses : Modern processors use split-transaction buses to maximize the efficiency of the progress of transactions since several transactions might be in progress simultaneously. With this optimization, it works against performance since it makes the coherence protocol complex because multiple memory operations can start without waiting for one to complete. Thus, the protocol should handle more transient states with possible timing problems.

Verification Complexity: In fact, verification of correctness of a hardware implementation of a coherence protocol is not easy. Simulation techniques can be helpful to some extent for testing specific scenarios although they cannot possibly guarantee the absence of errors like deadlock, livelock, or starvation for all such possible cases. The formal verification techniques, on the other hand verify all possible states. Correctness in hardware implementation is quite important. However, using formal verification techniques is again difficult due to complexity in multi-core systems.The management of concurrent, non-atomic state transitions is the main challenge in building multi-core hardware implementing the MESI cache coherence protocol, which often leads to transient states, race conditions, and deadlock. These adverse effects are even more significant with the potential for affecting the split-transaction buses and making this much harder to guarantee correct, efficient, and scalable cache coherency across multiple cores.

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

Literature Review

Cache coherence remains one of the most significant problems in multi-core systems and has led to extensive research and development over the years. Researchers have proposed and implemented novel coherence protocols, techniques, and verification methods. This chapter provides an overview of existing solutions, their limitations, and a literature review of state-of-the-art approaches in the field.

3.1 Existing Solutions in the Market

1. Snooping Protocols:

Overview: Snooping-based protocols, such as MESI (Modified, Exclusive, Shared, Invalid) and MOESI (Modified, Owned, Exclusive, Shared, Invalid), are widely used in commercial multi-core processors. These allow caches to observe, or "snoop," bus transactions to maintain coherence.

Disadvantages: These protocols result in increased bus traffic, causing bottlenecks, especially in core-dense systems. Snooping protocols typically scale well for small to moderate core counts but suffer from scalability issues at larger scales.

2. Directory-Based Protocols:

Overview: Directory-based protocols use a centralized or distributed directory to track the state of each cache line and its owner. Examples include the Scalable Coherence Protocol, Snoopy, and the directory-based MOESI protocol.

Disadvantages: While directory-based protocols reduce bus traffic and improve scalability, they introduce complexity in the directory structure, which can become a performance bottleneck. Latency in directory lookups is also a significant challenge.

3. Hybrid Protocols:

Overview: Hybrid protocols combine elements of snooping and directory-based protocols to balance their strengths and weaknesses. An example is the Hybrid Cache Coherence Protocol (HCCP), which uses snooping in local caches and a directory for inter-cache communication.

Disadvantages: Hybrid protocols are often complex and may introduce overhead in situations that require synchronization of state changes across caches.

4. Cache Coherence Over Networks:

Overview: System-on-Chip (SoC) architectures and distributed shared memory systems can ensure coherence via network protocols like the Cache Coherence Network.

Disadvantages: These solutions often suffer from latency issues due to network communication delays and are unsuitable for managing transient states that generate consistency conflicts.

3.2 Literature Review: State-of-the-Art Research in Cache Coherence

Recent research has advanced the understanding of key challenges and potential solutions in cache coherence for multi-core systems. Below are some notable contributions:

1. **"Cache Coherence Techniques for Multicore Processors" by Michael R. Marty (2008)**
   Discusses predictive techniques for improving cache coherence performance by reducing unnecessary coherence traffic.

2. **"Efficient Cache Coherence for Large-Scale Chip Multiprocessors" by Alberto Ros (2010)**
   Presents efficient primitives for synchronizing processes in large-scale chip multiprocessors.

3. **"An Efficient Cache Coherence Mechanism for Chip Multiprocessors" by Abdullah Kayi (2011)**
   Proposes novel coherence mechanisms for advanced multi-core architectures, with emphasis on emerging technologies.

4. **"Scalable Cache Coherence for Many-Core Architectures" by Alberto Ros (2011)**
   Introduces a network interface that allows the processor to launch and intercept coherence protocol packets efficiently.

5. **"Multicore Processors: Challenges, Opportunities, Emerging Trends" by Christian Märtin (2014)**
   A critical review of challenges in multicore processor design, highlighting trends and design decisions for multi-core systems.

6. **"A Performance Study of Cache Coherence Protocols in Multi-Core Processors" by Amit D. Joshi (2016)**

Offers a comparative analysis of snooping and directory protocols, identifying performance bottlenecks with increasing core counts.

7. **”Research on Cache Coherence Key Technology in Multi-Core Processor Systems” by Su Zhang (2016)**
   Examines literature on coherence protocols, exploring unexplored areas such as transient states and race conditions.

8. **”Analysis of Multi-Core Cache Coherence Protocols from Energy and Performance Perspectives” by Amit D. Joshi et al. (2018)**
   Focuses on analyzing existing cache coherence techniques, particularly in terms of energy efficiency and performance.

9. **”A Hardware Platform for Exploring Predictable Cache Coherence Protocols for Real-Time Multicores” by Zhuanhao Wu (2021)**
   Analyzes coherence solutions for embedded systems, emphasizing energy efficiency and performance.

10. **”Designing Predictable Cache Coherence Protocols for Multi-Core Real-Time Systems” by Anirudh Mohan Kaushik et al. (2022)**
    Addresses the challenge of enabling simultaneous and predictable access to shared data in multi-core systems, particularly for real-time applications.

3.3 Limitations of Existing Solutions

Although there are many solutions available in the market, and researchers are focusing on novel approaches, significant challenges remain:

- **Scalability:** Many existing protocols struggle to scale efficiently as core counts grow, leading to performance degradation due to increased latency.
- **Complexity:** Coherence protocols can become overly complex, making verification and maintenance difficult.
- **Transient States:** Many solutions fail to properly manage transient states, which can result in inconsistencies and race conditions, especially in high-frequency operations.

Chapter 4

Project Overview and Objectives

Overview and Goals

The project offers the implementation of a quad-core system that uses MESI protocol to tackle the problem of cache coherence. Our implementation 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 making contact with the L2. In this scenario, the methodology should be efficient and reliable.

Proposed Solution

The design is centered on a snooping MESI protocol for coordinating cache coherence. For each core, the design includes a private L1 cache, and to deal with the main memory interaction the system incorporates a shared L2 cache. The atomic nature of the bus ensures that only one transaction can take place at any given time; this simplifies coherence enforcement, ensuring optimal performance. [6].

Key Value Proposition

- **Configurability:** Our design allows parameterized core and cache setups, making it adaptable to various system.
- **Robust Coherence Protocol:** The MESI protocol provides reliable coherence management, preventing inconsistencies between cores and ensuring data integrity.
- **Balanced Performance:**: We truly believe in the effectiveness of correctness and coherence; however, our design also balances performance by using arbitration and write-back strategies [8, 9].

Project Objectives

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

Additionally, the Riscv-sodor framework provides a valuable foundation for implementing our quad-core architecture, demonstrating its adaptability for various system requirements [10].

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

Table 4.1: Summary of Read Test Cases for Cache Coherence System

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

Table 4.2: Summary of Write Test Cases for Cache Coherence System

Chapter 5

Project Development Methodology/Architecture

5.1 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 5.1: Parameter and Configuration

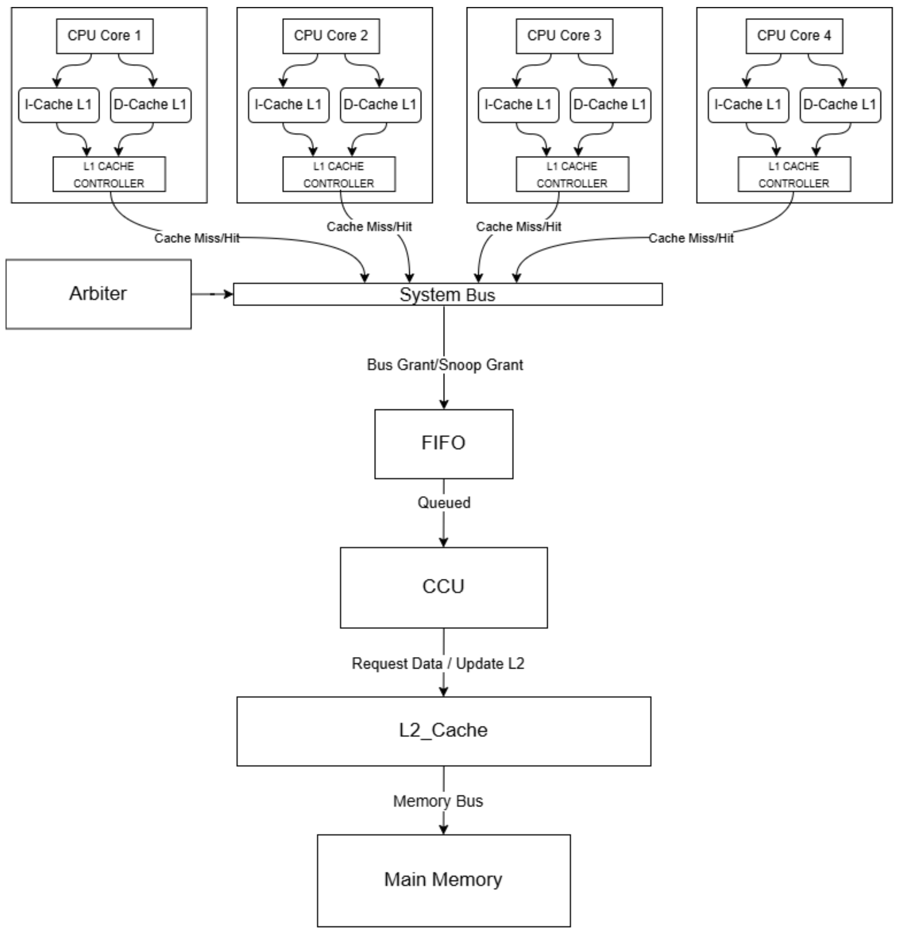


Figure 5.1: Block Diagram of the Quad-Core System Architecture

Tools and Technologies:

Hardware Description Language (HDL) like Verilog/SystemVerilog is used for the RTL design of all such components. Verification: UVM-based testbenches for functionality as well as for coherence. Simulation Tools: Vivado/QuestaSim and EDA Playground for simulating and debugging.

5.2 Summary of System Block Diagram

Interaction between the Core: Each core, in turn, initiates a memory read and write request to the L1 cache controller. Bus Arbitration: The arbiter grants priority wise bus access for the L1 cache controller. Cache Hierarchy: The L1 cache is private to the core and the L2 cache is shared between all cores. This improves access to shared memory and reduces misses. Memory Access: The main memory serves as the final data storage in the case of L2 cache misses. FIFO Buffers: These buffers manage pending requests when there are delays in accessing the bus or cache. This architecture ensures coherent and efficient access to data across all the cores of a machine, based on the MESI protocol for maintaining consistent states of memory within the multi-core environment. The system's performance can be further optimized, as suggested in prior research on automotive multi-core architectures [12].

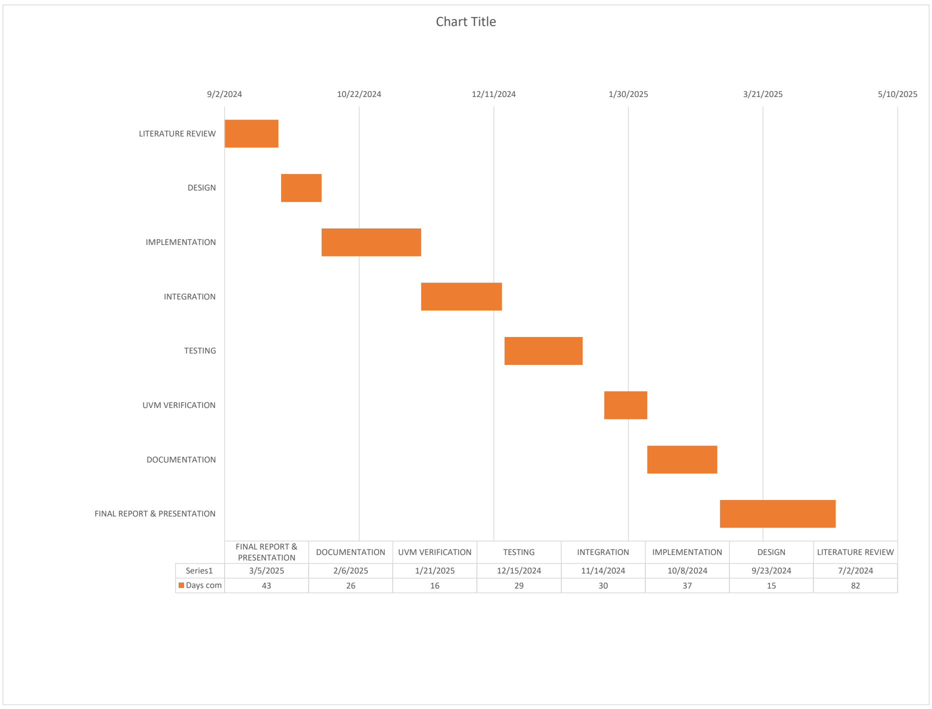
Chapter 6

Project Milestones and Deliverables

6.1 Gantt Chart

The project timeline highlights key milestones between October and March, during which coding and testing shall prevail predominantly. So, in October, the important activities are to include definition of the scope of the project and its objectives, literature review, building the system architecture, and coding the critical elements of multicore system. November shall be devoted to the continued implementation of the code as well as initial verification and integration tests. For December, it will be modules completion, improvement of the codes, and testing of individual modules. The Gantt chart presented in Figure 6.1: Gantt Chart of the Project provides a comprehensive overview of the project's timeline, illustrating the various phases of development. It serves as a crucial tool for tracking progress and ensuring that each task is completed within the designated time frame. The chart outlines key milestones, dependencies between tasks, and expected completion dates, allowing for efficient resource allocation and management throughout the project lifecycle.

Integration: January In the integration phase, the protocol for coherence functions would be ensured to operate as desired. A system level integration and simulation would be conducted. February The testing and optimization would continue, so there is refinement of the pseudo-LRU replacement policy, that the MESI protocol is correctly implemented. March Verification stress, and documentation will be done in order to ensure the project meets the desired performance, and correctness levels are achieved. This structured approach ensures gradual, iterative development in which both hardware and software components of the system will be well developed and tested by the project deadline.This methodology ensures that all components of the quad-core system architecture are developed cohesively, adhering to the planned schedule while allowing for adjustments as necessary. The iterative process reflected in the Gantt chart emphasizes the importance of continuous assessment and adaptation, which is essential for successful project execution.



	FINAL REPORT & PRESENTATION	DOCUMENTATION	UVM VERIFICATION	TESTING	INTEGRATION	IMPLEMENTATION	DESIGN	LITERATURE REVIEW
Series1	3/5/2025	2/6/2025	1/21/2025	12/15/2024	11/14/2024	10/8/2024	9/23/2024	7/2/2024
🟧 Days com	43	26	16	29	30	37	15	82

Table 6.1: Gantt Chart of the Project

Chapter 7

Block Diagram

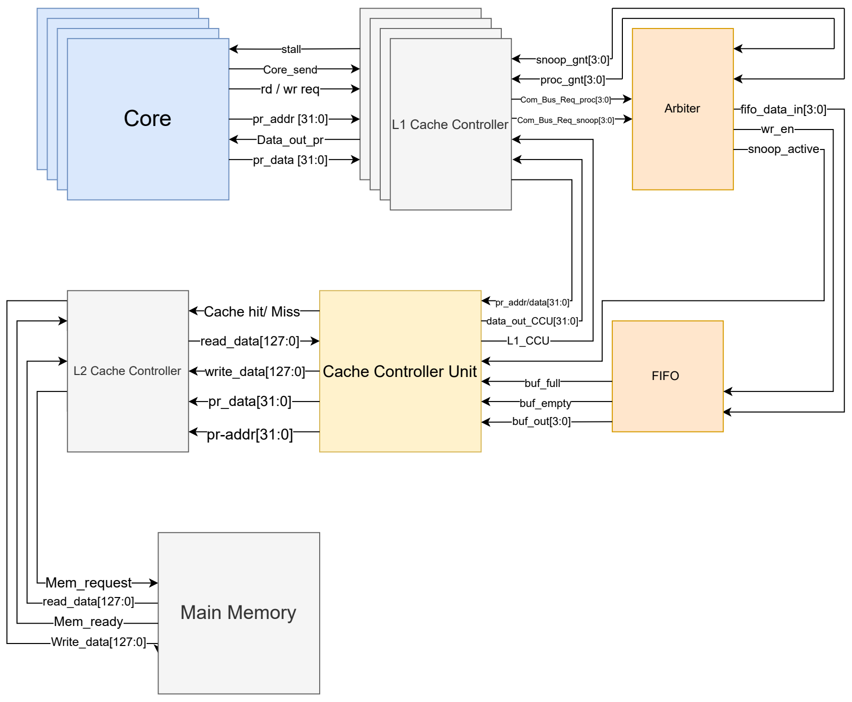


Figure 7.1: Block Diagram

A multicore processor system with a memory hierarchy is depicted in Figure 7.1. Let's dissect the essential elements:

1. **Core(s):** A number of processor cores that carry out instructions are indicated in blue.
2. **L1 Cache Controller:** Each core is directly attached to the first level cache controller.
3. **L2 Cache Controller:** Cores share a second-level cache controller.
4. **Primary Memory:** The primary memory (RAM) of the system.
5. **Cache Controller Unit (CCU):** Coordinates the L1 and L2 caches.
6. **Arbiter:** Prevents disputes by controlling access requests from several cores.
7. **FIFO:** A memory request queue that uses a first-in, first-out buffer.

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 CCU to the L2 cache.
3. The request moves to Main Memory in the event of an L2 cache miss.
4. When several cores request resources, the Arbiter makes sure that access is ordered.

Control signals that help keep cache coherency 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 8

Flow Chart

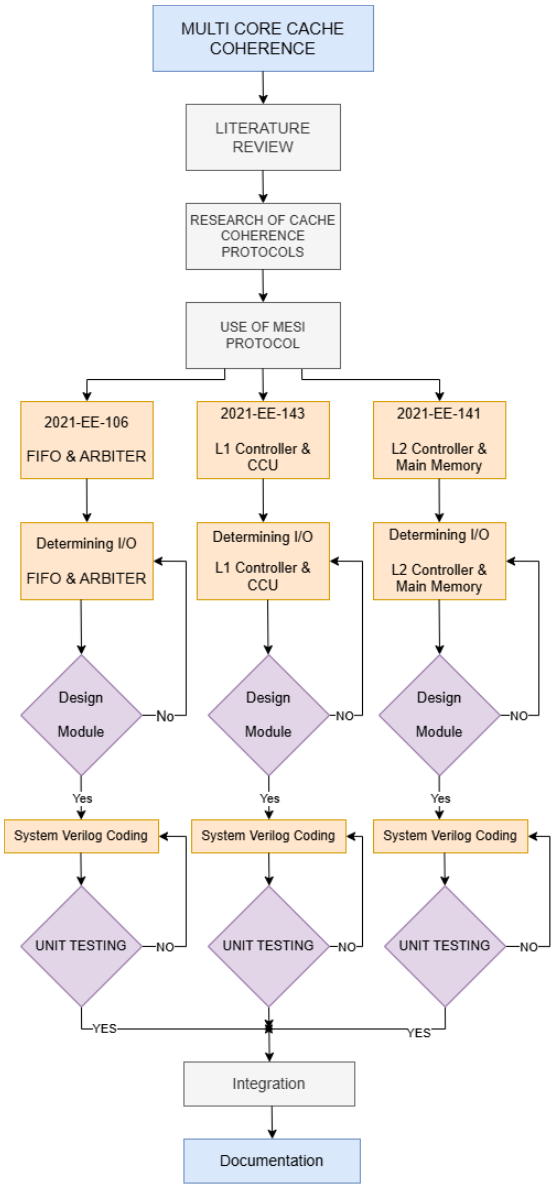


Figure 8.1: Flowchart

Chapter 9

Work Division

1. Literature Review and Research (Common Task):

This phase involves understanding cache coherence protocols and selecting the MESI protocol, which serves as a foundational task for the entire team. It ensures that all members have a clear understanding of the core concepts of the project.

2. Module 1: FIFO & Arbiter (2021-EE-106)

Task Division:

- **Specify I/O:** Team Member 1 (2021-EE-106) will enumerate the input and output signals required for the components of the FIFO and the Arbiter.
- **Design Module:** Design the architecture and logic of the FIFO and Arbiter, ensuring adherence to system requirements.
- **System Verilog Coding:** Implement the design in System Verilog, including all modules and their submodules.
- **Unit Testing:** Develop test cases for the functionality of both the FIFO and Arbiter to ensure they meet project specifications.

Objective: Ensure core requests are served correctly using the FIFO, and manage bus contention effectively.

3. Module 2: L1 Controller & Cache Control Unit (CCU) (2021-EE-143)

Task Breakdown:

- **Determining I/O:** Define inputs and outputs for the L1 cache controller and CCU, focusing on maintaining cache coherence using the MESI protocol.
- **Design Module:** Develop the L1 cache controller and CCU, ensuring compliance with the MESI protocol.

- **System Verilog Coding:** Implement the logic for the L1 cache controller and CCU in System Verilog, testing for inter-core communication.
- **Unit Testing:** Write test cases for the L1 cache controller and CCU to verify accurate execution of cache operations (read/write/invalidation).

Module Objective: Ensure proper execution of cache operations and coherence between cores.

4. Module 3: L2 Controller & Main Memory (2021-EE-141)

Task Breakdown:

- **Determining I/O:** Identify inputs and outputs for the shared L2 cache and its interface with the main memory.
- **Design Module:** Create the L2 controller that serves all cores and interfaces with the main memory.
- **System Verilog Coding:** Implement the L2 controller in System Verilog to manage data writing to and fetching from main memory.
- **Unit Testing:** Develop test cases that ensure proper interaction between the L2 controller, cores, and main memory.

Task Objective: Verify that the shared L2 cache operates correctly and supports the overall cache coherence scheme.

5. Collaboration with Team on Integration and Documentation

During the testing and validation phases of individual modules (FIFO & Arbiter, L1 Controller & CCU, L2 Controller & Main Memory), the team will collaborate to integrate these components into a cohesive system. Finally, comprehensive documentation will be created to ensure each component is thoroughly recorded, facilitating future assessments and project continuity.

Chapter 10

Costing

Since the project was mainly software-based, it did not require component costing. It was significantly about design and simulation, which an environment like verilog, UVM, and Simulation software such as EDA Playground, ModelSim, or Vivado might handle through some free or academic license versions. The development or testing of the project would not require physical hardware components; therefore, it reduced the budgeting of hardware components.

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.

[12] S. Schliecker, J. Rox, M. Negrean, K. Richter, M. Jersak, and R. Ernst. System level performance analysis for real-time automotive multicore and network architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2009.