POLITECNICO DI TORINO

Master´s Degree in Computer Engineering

Master Degree Thesis

UVM verification for Multi-core architecture RISC-V

Supervisor:

Prof. Ernesto SANCHEZ

Co-Supervisor:

Prof. Annachiara RUOSPO

Candidate

Ismiana QOSE

July 2025

Abstract

Since its release in 2011, the RISC-V Instruction Set Architecture (ISA) has gained widespread growth thanks to its open-source nature and features. RISC-V has a lot of free core designs that cover a wide range of applications, including embedded systems, virtual and augmented reality (VR/AR), artificial intelligence (AI), and the Internet of Things (IoT).[1] As RISC-V cores have become more common, this has led to a necessity for multi-core systems that are both resource-efficient and high-performing. Even though many approaches have been introduced for integrating a multi-core system on traditional architectures, it still remains difficult to create an effective ecosystem that makes use of all RISC-V's characteristics. This is mostly due to the fact that the majority of open cores are released as a single core without cache coherence logic, which needs costly development and design work to fix.

This paper describes the architectural changes necessary to upgrade the CVA6 RISC-V processor from a single to a dual-core configuration. The CVA6 is programmed in SystemVerilog and has several configuration options. It introduces a scalable methodology for integrating two CVA6 (Ariane) RISC-V cores within a shared system-on-chip (SoC) environment, emphasizing efficient memory access and resource sharing. Both cores are connected to a unified memory subsystem via an interconnect mechanism with arbitration logic. To handle inter-core coherency and ensure proper transaction management when both cores access overlapping memory regions, an invalidation mechanism is employed.

The verification effort leverages the Universal Verification Methodology (UVM) to ensure the correctness and robustness of the dual core design. The UVM environment is extended to support dual-core scenarios, including updates to agents, monitors, and scoreboards for handling transactions from multiple cores. Arbitration and memory access are validated through UVM sequences, while coverage metrics ensure thorough testing suite. It is implemented to measure the completeness of transaction scenarios. Code coverage ensures that all lines of code and conditional branches are exercised during simulation.

The design is validated through simulation, waveform analysis, log inspection, and UVM-based verification, demonstrating correct operation, successful arbitration, and effective transaction handling. This approach highlights the effectiveness of arbitration methodology, invalidation mechanisms, and UVM-based verification for simple multi-core RISC-V architectures, providing a foundation for robust and adaptable designs.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor Prof. Ernesto Sanchez and my co-supervisor Prof. Ruospo Annachiara for their guidance, continuous support, and insightful advice throughout the course of my thesis. Their expertise and dedication have been instrumental in shaping the direction of this research and overcoming numerous challenges along the way. I am also especially grateful to Behnam Farnaghinejad for his patience, encouragement, and for always challenging me to think critically and pursue excellence in my work.It has been a privilege to work under their supervision and I am forever thankful for the inspiration and support they have provided throughout this journey.

"Ismiana"

Table of Contents

List of Tables

List of Figures

Acronyms

UVM
Universal verification Method

PMP
Physical Memory Protection

ISA
Instruction Set Architecture

AXI4
Advanced eXtensible Interface

UVC
Universal Verification Components

ASIC
Application-Specific Integrated Circuit

FPGA
Field-Programmable Gate Array

CSR
Control and Status Registers

RAS
Return Address Stack

BTB
Branch Target Buffer

ID	Instruction Decode
OOB	Out-of-Order Writeback
ALU	Arithmetic Logic Unit
RVFI	RISC-V Formal Interface
DUT	Device Under Test
FU	Functional Unit
PTW	Page Table Walk
AXI	Advanced eXtensible Interface
SoC	System on Chip

Chapter 1

INTRODUCTION

1.1 Background

The growing demand for high-performance and energy-efficient computing systems has driven the widespread adoption of multi-core processor architectures. These methodologies enable parallel processing, improve computational throughput, and provide better resource utilization for modern software applications.

Despite being intended as a single-core processor, the CVA6 core is a high-performance 64-bit RISC-V processor that can be extended to dual-core configurations because of its open-source nature. This features have made it a leading platform on fields related to research, experiments, and deployment of scalable embedded systems and custom SoC designs. The actual dual-core scenario often rely on designs with limited flexibility and high development costs.

Projects such as CVA6 within the open-source RISC-V system, can simplify hardware development and can motivate collaborative innovation. Modifying the CVA6 core for dual-core support goes in line with these trends by offering a configurable and cost-effective solution for high-performance computing applications.

Dual-core processors, which integrate two independent processing units on a single chip, have been representing a significant milestone in this progression. The limitations of single-core processors related to thermal constraints and diminishing returns from increasing clock speeds have been addressed through his methodology. They have become a fundamental element for modern computations technology and for powering a wide range of applications starting from personal devices up to large-scale enterprise servers.

Multi-core systems offer some considerable benefits in terms of metrics related to performance, scalability, and flexibility. They enable advanced features in flexible communication protocols and in cache coherence mechanisms with the purpose of maintaining data consistency across cores.

However this benefits come with some challenges in terms of adoption of multi-core architectures. When the number of cores in the system is increased the design and verification complexity increases significantly, requiring advanced tools and methodologies to reach correctness and performance specifications. Furthermore, this architecture experiences additional area and power overheads, which must be carefully managed to preserve the efficiency. Nevertheless, the advantages of multi-core systems render them as essential components of modern computing platforms, with innovation in both hardware and software design.[2]

1.1.1 Analysis of Comparable Research

The work presented in this paper is focused on modifying the UVM environment to adapt it with dual-core scenario rather than modifying the RTL design. Two CVA6 cores are integrated in the SoC. Communication between cores and memory is done thorugh a multiplexer logic that incorporates together features such as arbitration and response routing. The idea is to focus on building a smarter testing environment that can handle the complexities of multiple cores, rather than directly tweaking the hardware.

Multi-core systems struggle a lot due to the demand for effective interaction between cores, shared memory management, and proper interrupt handling to avoid conflicts. Debugging process when issues arise in such systems can become highly challenging. Modifications required in the hardware (RTL) to address these problems are often time consuming, because they require extensive re-synthesis and multiple design iterations. Moreover, changes to the RTL can jeopardize the design flow, and can result in further verification effort and stability issues.

Below in Table 1.1 is represented an overview of existing RISC-V cores. As seen, most of these cores are released as single-core configurations without a built-in Cache Coherence Logic (CCL). This limitation forces platform developers to independently design and implement the CCL when configuring multi-core systems.

In some cases, the absence of accessible or readable RTL code further complicates this task, making it impossible to integrate CCL. On the other hand, some RISC-V cores support multi-core configurations with CCL but are limited by fixed core counts. Developers aiming for scalable or heterogeneous multi-core systems often face the challenge of implementing their own CCL, making platform development complex and resource-intensive.

Supplier	**Core Name**	**# of Cores**	**Cache**	**CCL**
Andes	A25, D25F, N22	1	Yes	No
Andes	A25MP, AX25MP	1∼4	Yes	Yes
ETH	Ariane, Zero-riscy	1	Yes	No
CloudBEAR	BI-350, BM-310	1∼4	Yes	No
Codasip	BK3, BK5, BK7	1	Yes	No
Darklife	DarkRISCV	1	No	No
Bob Hu	Hummingbird E200	1	Yes	No
FPGA Cores	Instant SoC	1	No	No
Cornell	Lizard	1	Yes	No
LambdaConcept	Minerva	1	Yes	No
SiFive	Rocket	Multi	Yes	Yes
Western Digital	SweRV EH1	1	Yes	No
IIT Madras	Shakti-Eclass, 1class	1	Yes	No
Roa Logic	RV12	1	Yes	No

Table 1.1: Features of the existing RISC-V cores

1.1.2 Modular multi-core methodology

Modern computing systems tend to have a higher performance and to deliver energy efficiently. To achieve this they are relying on adaption to multi-core architectures. Open-source platforms are now crucial for education and research purposes, providing rapid prototyping and innovation.

The OpenPiton project, developed by Jonathan Balkind, Michael McKeown, Yaosheng Fu, and Tri Nguyen, proposes a rigorous open-source many core research framework.[2] OpenPiton's modular structure, scalability, and extensive documentation have established it as a cornerstone for research reasons. It provides both advanced experiments and reproducible outcomes.

Taking leverage from OpenPiton, the dual-core implementation follows similar design philosophies:

- **Open-source hardware principles**

- **Modular, parameterized architecture**
- **Emphasis on cache coherence and shared memory communication**
- **Arbitrated on-chip interconnects for safe memory sharing**

OpenPiton targets many core scalability (hundreds of cores) and supports SPARC ISA. On the other side the focus of our work is on a dual-core RISC-V system with the following variations:

- **Core Architecture:**

OpenPiton is based on SPARC structure and aims scalability while Dual-Core focuses on integration of two instances of the open-source CVA6 RISC-V core.

- **System Size and Complexity:**

OpenPiton is designed to support large-scaled systems while this work provides a simpler dual-core setup environment, which is ideal for educational use, verification, and exploration of coherence protocols.

- **Cache Coherence:**

OpenPiton implements a directory-based coherence protocol for maintaining coherence among many cores while this approach uses a simple invalidation-based coherence logic for dual-core setup.

- **Communication Fabric:**

OpenPiton uses a network-on-chip (NoC) for inter-tile communication while this approach uses a shared subsystem with arbitration features.

- **Research Focus:**

OpenPiton points out scalability, heterogeneity, and research extensibility while this approach emphasizes clarity, testability, and serves as a minimal starting point towards multi-core cache coherence research.

Summary

Inspired from OpenPiton, this dual-core architecture demonstrates how open-source, modular, and scalable design principles can be adapted and simplified but yet providing support for educational and research goals in the context of RISC-V. It enables hands-on learning, rapid iteration, and a clear path toward more complex many core systems.

1.2 Motivation

The RISC-V system is quite popular due to its open-source and extensible features, and has became an attractive platform for hardware innovation. However, some of the existing multi-core implementations are either restricted or complex, making them not accessible to academics and developers. The motivation to adapt CVA6 project to support a multi-core configuration is driven by several factors:

1. Parallel Computing Needs: Many modern applications, starting with data analytics, machine learning, and real-time systems, require multi-threaded processing for better performance.

2. Open-Source Innovation: Making use of the open-source nature of CVA6 provides innovation and collaboration within the research and development community.

3. RISC-V Ecosystem Growth: Optimization of multi-core capabilities for RISC-V processors aligns with the growing interest in RISC-V as a flexible and scalable hardware platform.

4. Cost Efficiency: Developing open-source multi-core architectures reduces dependency on high cost designs and offers a customized solution for a lot of applications.

5. Research and Education: The availability of this platform gives some beneficial resource and studies for academic institutions and research groups that analyze and work on parallel processing, hardware design, and system software development.

The motivation to implement this architecture came as the need to build a simple, significant, efficient, and yet functional multi-core system. This approach can demonstrate inter-core coherence in terms of access write channel and fair arbitration. It can support 2 up to 4 cores. However it has its own limitations,when increasing the number of cores is increased the design and verification complexity, as well as arbitration overhead and contention. So it is not compatible with many core systems scalable from 10s to 100s of cores.

1.3 Objectives

The main objectives of this project are related to expansion of CVA6 RISC-V system to support dual-core configurations, to enable shared memory access and to improve computational efficiency. To accomplish this, several key technical and design goals must be reached:

1. Develop a scalable hardware interconnect to seamlessly integrate both CVA6 cores, in order to facilitate communication and resource sharing among cores.

2. **A**chieve cache coherence goals among both CVA6 cores by continuously observing only the AXI AW (write address) channel to ensure that each core is properly informed of write operations by the other peer. This allows timely invalidation of stale cache lines without introducing unnecessary broadcasts or complexity.

3. **B**uilt a system where both targeted cores can operate beneficently together with the shared memory component and support collaborative communication and data exchange.

4. **P**rovide a robust multi-core system that establishes systematic verification for advanced RISC-V research purposes.

5. **G**uarantee fairness and cyclic access to shared memory among both cores with equal possibility to access the memory and systematically rotating access priority, so that no core is starved regardless of request patterns.

By accomplishing these objectives, this project seeks to update the CVA6 core into a versatile and scalable multi-core processor, thus it will contribute to the growing of RISC-V ecosystem and will address the demand for high-performance computing solutions.

Chapter 2

Literature Review

2.1 RISC-V Architecture

The RISC-V architecture is a modern, open-standard Instruction Set Architecture (ISA) designed to be simple, scalable, and extensible. It supports a wide range of applications starting from high-performance computers continuing to embedded systems. Because of the architecture's high degree of modularity, developers may create unique extensions while still adhering to the underlying ISA.

It is adaptable for different applications because of its versatile nature. The architecture covers both 32-bit (RV32) and 64-bit (RV64) address spaces,as well as 128-bit (RV128). This flexibility makes possible for it to be employed in a wide range of computing environments, from low-power embedded devices to powerful server-grade processors. [1]

2.2 RISC-V Memory Ordering Specifications

Memory consistency models vary from weak to strong. Weak memory models are able to let more hardware implementation flexibility and deliver better performance in terms of watt, power, scalability, and hardware verification overheads compared to the strong models. Strong models provide simpler programming models, but with the cost of facing more restrictions regarding (non-speculative) hardware optimizations that can be performed in the pipeline and in the memory system. This results in higher cost in terms of power, area overhead, and verification burden. [3]

RISC-V uses the RVWMO (RISC-V Weak Memory Ordering) memory model, which puts it in between the two extremes of the memory model spectrum. The RVWMO memory model allows constructing simple implementations, aggressive implementations, implementations embedded deeply inside a much larger system and subject to complex memory system interactions, all while simultaneously being strong enough to support programming language memory models at high performance.

RISC-V defines its memory model (RVWMO) in the context of 3 orderings:

- **Program Order:** It represents a sequence of instructions executed by a single (cores) hart before any out-of-order effect imposed by hardware or compiler. They are written in the program and can be affected if processor reorders them for optimization purposes.
- **Global Memory Order:** It represents the order in which loads and stores perform. The formal memory model has moved away from specifications built around the concept of performing, but the idea is still useful for implementing informal intuition. A load is said to have performed when its return value is determined. A store is said to have performed not when it has executed inside the pipeline, but rather only when its value has been propagated to globally visible memory. In this sense, the global memory order also represents the contribution of the coherence protocol and/or the rest of the memory system to interleave the (possibly reordered) memory accesses being issued by each hart into a single total order agreed upon by all harts.
- **Preserved Program Order:** It represents the subset of program order that must be respected within the global memory order. Conceptually, events from the same hart that are ordered by preserved program order must appear in that order from the perspective of other harts and/or observers. Events from the same hart that are not ordered by preserved program order, on the other hand, may appear reordered from the perspective of other harts and/or observers. [3]

Cores in RISC-V architecture have their own private cache, and because of this it is important to ensure that they are all observing consistent values from shared memory subsystem. As a result it is essential to implement cache coherence protocols which will monitor the status of each cache line. When implementing such mechanism is crucial not to violate the specification defined by the system, and to ensure that all required cache updates and invalidations are performed correctly respecting global memory order.

2.3 CVA6 Core

CVA6 (formerly known as Ariane) is an open-source, 64-bit, in-order RISC-V core developed by the OpenHW Group and initially contributed by ETH Zurich.[4]

It operates with RV64GC standard (64-bit base ISA with general-purpose and compressed extensions) and is totally in sync with the RISC-V ISA specification.

It is made for high-performance embedded applications. Although individual cores may target a particular implementation technology, the CVA6 targets both ASIC and FPGA implementations. SystemVerilog is used to write the highly parameterizable CVA6. For instance, options allow you to adjust the ILEN to either 32 or 64 bits and enable or deactivate floating point capability.[4]

2.3.1 Key Features

- **RISC-V Compliance:** This feature is adapted to support the RV64GC (64-bit, general-purpose, and compressed instructions) standard, and implements privilege modes such as User, Supervisor, and Machine.
- **High-Performance In-Order Design:** It is a s-stage in-order pipeline useful for instruction execution and is optimized for better performance and power in embedded systems.
- **Memory and Cache Support:** The architecture includes L1 Data and Instruction Cache which are configurable for performance optimization and includes an optional L2 Cache as well to improve memory latency for complex workloads.
- **MMU (Memory Management Unit):** It offers a virtual memory support with page-based memory mapping, which is also adaptive with Linux or other operating systems.
- **Floating-Point and Vector Extensions:** It supports floating-point operations by including F and D extensions and is additionally prepared for further extension with RISC-V vector processing capabilities.
- **AXI4 Interconnect Support:** One important feature of this architecture is also the support of AXI4 bus interface used for efficient implementation with memory controllers and other SoC components.
- **Interrupt Management:** It is totally compilant with RISC-V interrupt specifications for reliable handling of software and hardware interrupts.
- **Open-Source and Customizable:** It's open-source features, allow it to be relevant for academic research and commercial applications.
- **Toolchain Compatibility:** It is compatible with many toolchains such as RISC-V GCC, LLVM, and other software tools. Can also support Linux and other real-time operating systems (RTOS).

- **Synthesized in FPGAs and ASICs:** It is integrated on FPGA boards for testing and prototyping purposes and is integrated in ASIC implementations for tape-out purposes. [4]

2.4 Multi-Core Approach

Multi-core architectures have become recently a fundamental component responsible for accomplishing the demand for higher performance and energy efficiency computing systems. They are increasing and improving concurrent execution of many instruction set by focusing on the throughput and outcome of parallel workloads. The RISC-V ISA is a flexible, open alternative to proprietary ISAs, and is widely adopted in research and commercial field. The CVA6 (formerly Ariane) core is a high-performance, Linux-capable, in-order 64-bit RISC-V processor implemented in SystemVerilog and Chisel. While CVA6 is architecturally sophisticated, it is designed as a single-core implementation, limiting its utility in evaluating and developing multi-threaded or parallel systems. Multi-core extension of CVA6 accomplish the needs of both academic research and industrial research by providing:

- Studies and researches on inter-core communication, synchronization, and scalability,
- Development of multi-core RISC-V software stacks (e.g., SMP Linux),
- Custom hardware-software co-design for domain-specific applications.

Chapter 3

Methodology

The goal of this work is to implement a dual-core RISC-V system using two CVA6 (Ariane) cores with shared memory access support. The methodology follows a modular, parameterized approach, that ensures scalability and ease of integration for future use as well. In the design has been incorporated interconnects, a custom arbitration (mux) module, and a dedicated invalidation filter to verify that each core is notified of writes by the other peers.

3.1 Experimental Setup

The first step of this approach is integration of a dual-core CVA6 system in a controlled simulation environment. Each core has been configured with a unique identifier and independent interfaces to facilitate the concurrent communication with memory. Tracking and monitoring infrastructure have provided detailed information regarding data execution.

One of the critical components of this design is the interconnect mechanism that is able to control simultaneous memory requests from both cores. It features the arbitration logic of a round-robin scheme and prepends core identifiers to AXI transaction IDs, to make sure that responses are correctly routed back to the originating core. This mechanism is essential for maintaining transaction integrity and traceability in a shared memory environment.

To address potential data hazards being caused by concurrent memory accesses, the system incorporates a mechanism related to invalidation logic. It continuously monitors the addresses and valid signals of incoming requests from both cores. It ensures that each core is informed of memory accesses by its peers, thereby maintaining a consistent and up-to-date view of shared memory resources. It can be used to trigger invalidation logic, such as stalling one core or invoking cache coherence protocols. The platform is equipped with RISC-V Formal Interface

(RVFI) probes and UVM agents, to provide detailed monitoring of instruction execution, memory transactions, and protocol compliance.

3.2 UVM Approach

This section describes the structured approach used to modify the CVA6 RISC-V core from a single-core architecture to a dual-core configuration. On the focus have been modifications needed for the current version of UVM (Universal-Verification-Methodology) environment regarding CVA6. Components such as tesbench development, verification strategy, scope of test and UVM environment itself are the highlights of the approach. In order to give a detailed description of the development process , below has been analyzed each phase.

Initially, it is important and convenient to understand the limits of the existing single-core UVM environment and to state the requirements for this implementation. The design needs to be scalable, to effectively support dual-core verification, to test data consistency across it and to verify core communication and synchronization mechanisms.

3.3 UVM Environment

3.3.1 UVM Testbench Adaptation

The current single-core environment is extended to support multi-core verification through the following procedures:

1. **Testbench Hierarchy Modification:** The testbench structure is adapted to integrate and manage two cores under verification.
2. **Environment Configuration:** Parametrization is used as a technique to configure dynamically the number of cores under test.
3. **UVC (Universal Verification Components)** Enhancements: The existing UVCs are extended and reused to handle dual-core transactions and responses.

3.3.2 CVA6 Verification ENV

Below in the Figure 3.1 is demonstrated a detailed step-by-step representation of modifications of UVM environment to support our dual-core verification strategy. On this representation are involved components such as testbench adaptation, stimulus generation, verification strategy development, and extensive validation to ensure a robustness and scalability. Then will be discussed the outcomes and analysis of the verification process, highlighting key results.

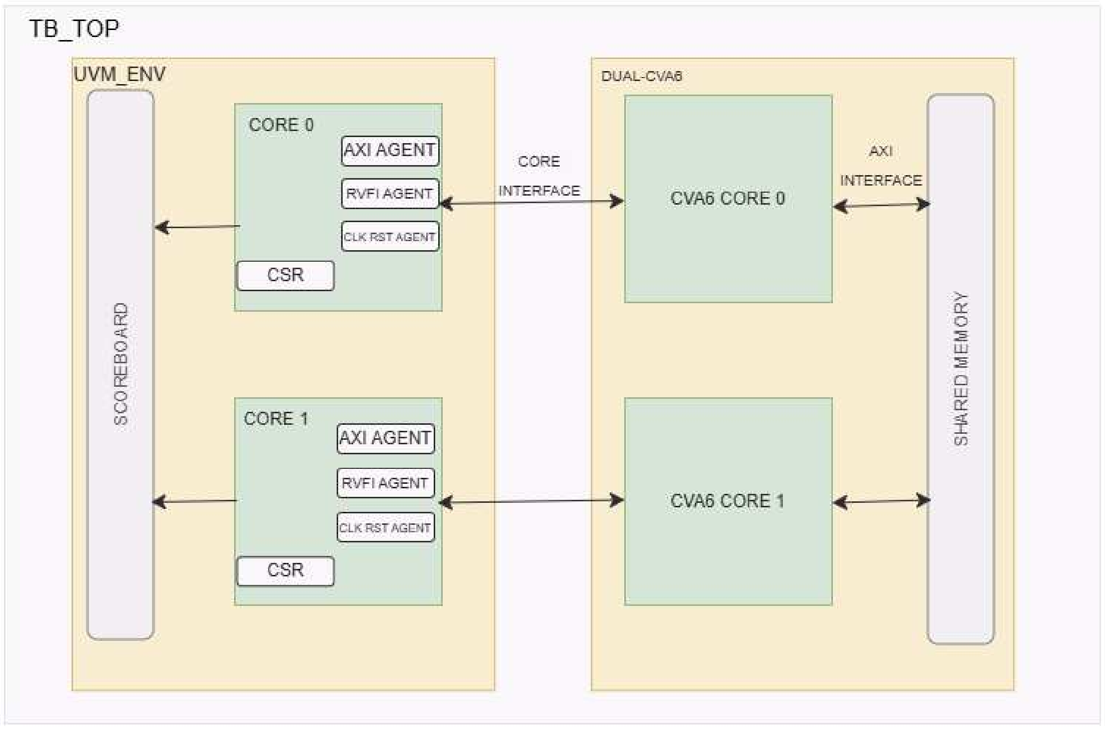


Figure 3.1: CVA6 Verification Methodology

Key Components in Detail

1. **CVA6 Environment**

   This environment is responsible to verify both cores and their interactions with shared memory. Key components include: **Core-Specific Blocks** Each core (Core0, Core1, etc.) is considered as a separate unit within the environment. For every core, the following agents and components are defined:

   Clkreset Agent: It is an agent that handles clocking and reset logic on the verification environment.

   AXI Agent: It is an agent that verifies memory transactions and AXI interface behavior of the cores.

   RVFI Agent: It is an agent that verifies the RISC-V Formal Interface (RVFI), which duty is to monitor each core's action for compliance and correctness purposes.

   CSRs: Control and Status Registers (CSRs) are verified for providing information regarding correct read/write behavior of the system.

2. **Scoreboard**

The scoreboard has an important role in collecting and verifying outcomes from cores and their agents. It is responsible to compare this actual information with the expected outputs to ensure that the system is functionaly correct as required. By doing this check it verifies data coherency across all cores. Additionally, it may track specific metrics such as coverage, pass/fail rates, and performance statistics to provide insights into the verification process.

3. **Device under Test**

   The DUT (Design Under Test) describes what is eesential to be processed. DUT in this scenario includes two CVA6 cores that independently process instructions while they are sharing resources. The AXI interface makes the communication between the cores and the shared memory easier, by ensuring efficient data transfer.

   On the other side shared memory subsystem and controller coordinate resources access across cores as required. The environment uses the AXI interface to simulate realistic memory accesses, enabling accurate verification of multi-core operations.

3.4 Coverage Metric

One of the most used metric in software and hardware verification is code coverage. It measures the degree at which the test suite executes the implementation under test. It identifies which parts and how much percentage of the source code have been executed during simulation or testing, thereby it helps in determining the comprehensiveness and effectiveness of the process.

Within functional verification, code coverage is particularly helpful in identifying parts of the design that have not been tested since this areas can potentially cause hidden bugs. It highlights areas of the test scenarios which are incomplete, in order to assist verification engineers in remodeling their approach and achieving higher accuracy in terms of correctness of the work.

During this process is evaluated code coverage with the following metrics:

- **Line Coverage:** This metric is able to track and identify whether each line of the code is executed during simulation. It provides a basic but yet efficient way of showing which parts of the design have been executed by test scenarios.

- **Toggle Coverage:** This type of code coverage metric can measure whether each bit of a signal has transitioned from 0 to 1 or from 1 to 0 at least once during simulation. Toggle coverage plays a crucial role in hardware verification because it reveals if the signal transition within registers, wires, and internal logic has been fully experienced.

- **Condition Coverage:** Condition or expression coverage is a metric that evaluates if all boolean sub-expressions within control statements (e.g., if, case) have been independently executed to both states either true or false. Compared to branch-coverage this is more fine-grained and provides details regarding the completeness of decision logic testing.

3.5 Performance Evaluation and Validation

3.5.1 Functional Validation and Debug/Analysis

The environment is modified to validate multi-core functionality, by offering a controlled setup for test suite and various aspects of the system. Extensive tests are carried out to check the cores communication, and to ensure coherency within them. Thereby this can guarantee the accuracy and reliability of the multi-core architecture.

Waveform viewers have been utilized as a way to analyze interactions and to check debug issues. They provide valuable insights into system behavior if it is as expected. Additionally, test logs are examined and studied in details to identify and resolve potential failures on the system.

3.5.2 Coverage Analysis and Iterative Optimization

Coverage reports are generated and analyzed under all scenarios in order to check if everything have been adequately tested. This reports tend to address any coverage bottleneck.

Based on the outcome of verification process, iterative optimizations are carried out. These includes a refinement of the testbench to improve the performance of UVM environment, expanding coverage points to capture more scenarios, and to optimize test execution time for multi-core simulations.

Chapter 4

Design and Implementation

4.1 CVA6 Core Overview

The CVA6 core's goal is to run an entire operating system OS at a defined speed and IPC. The core has a 6-stage pipelined architecture to get the required speed. The CPU has a scoreboard that should conceal delay to the data RAM (cache) by sending data-independent instructions in order to raise the IPC.

Accesses to the data RAM (or L1 data cache) have a greater delay of three cycles on a hit than those to the instruction RAM (or L1 instruction cache), which has an access latency of one cycle on a hit.[4]

The diagram on Figure 4.5 provided by ARIANE offers an overview of the CVA6 core architecture, breaking down its key components and pipeline stages. [4] In the below subchapter is given a detailed explanation.

4.2 System Design

The modifications in the system consist on enabling efficient, coherent, and scalable features using two CVA6 (Ariane) RISC-V cores. A seen in Figure 4.2 it is able to ensure correct operation under concurrent memory access scenarios by using industry-standard AXI4 interfaces, a custom arbitration scheme, and a coherency management unit. The bellow subsections explain each major subsystem and their integration.

4.2.1 Dual-Core CVA6 Integration

Two CVA6 cores have been integrated in this system, each instantiated within the top-level wrapper module. On each of them as demonstrated in Figure 4.1 is running a different program and are configured with unique hardware thread

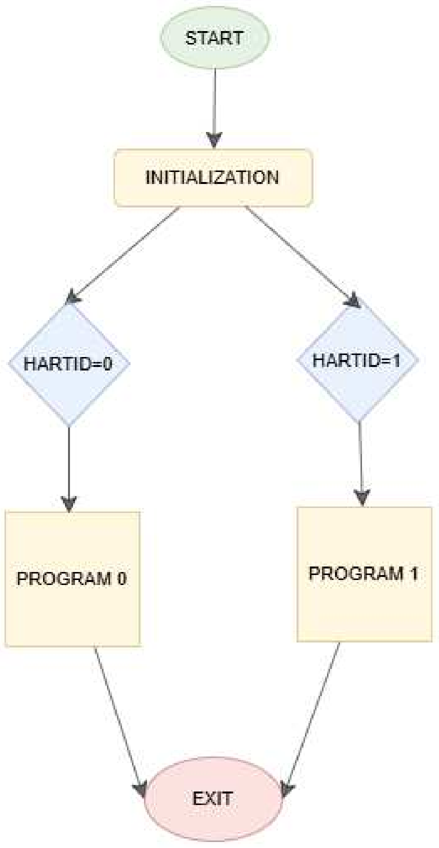


Figure 4.1: Program Execution

identifiers (ID is 0 for Core 0, ID is 1 for Core 1), in order to activate independent execution contexts. They are connected to shared interrupt, debug, and boot address signals, to provide synchronized startup and coordinated exception handling. The current implementation facilitates future expansion to more cores or usage with heterogeneous processing elements.

Each core uses an AXI master interface for memory transactions, as well as signals for debug and interrupt. The design supports full compliance with the RISC-V privileged architecture, allowing for robust multi-threaded and multi-core software execution.

4.2.2 Interconnect and Arbitration

The cores are communicating with shared memory subsystem with the help of the multiplexer logic, which is connected to each core´s AXI master interface. The multiplexer is applying a round-robin arbitration scheme. This approach is a

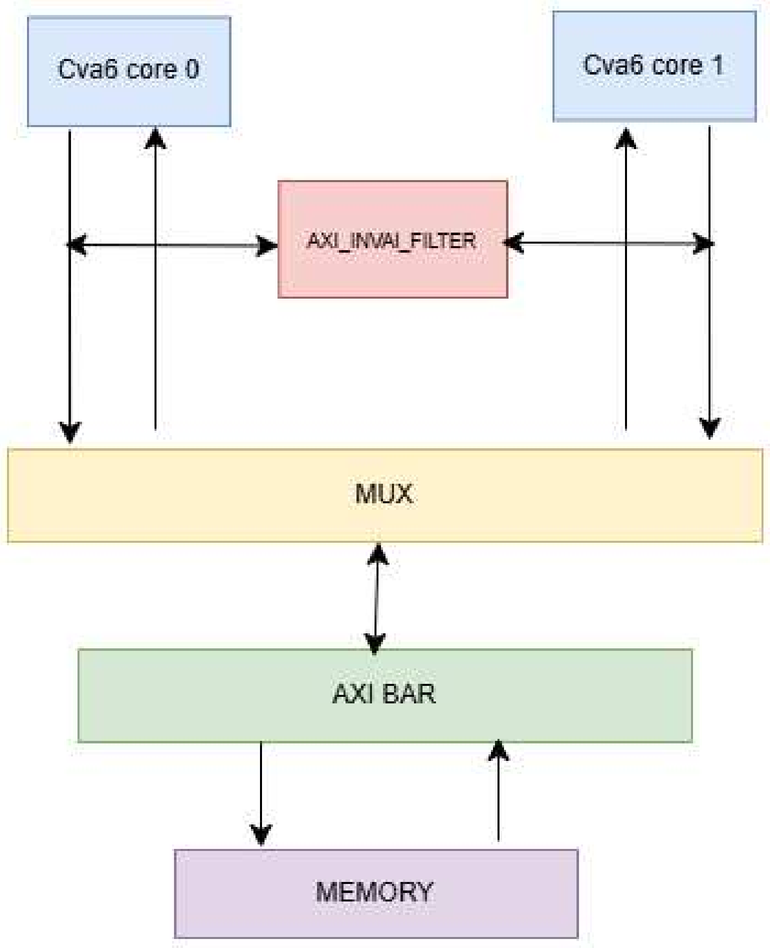


Figure 4.2: Dual-Core System

parameterized, scalable and capable to arbitrate requests from multiple masters. In this implementation it supports two slave ports (one per core). Memory access is granted to the cores in a cyclic order following the logic of the arbiter, regardless of the request urgency or core priority.
Key features of the arbitration logic include:

- **ID extension:** Transaction IDs are extended with a core index, to ensure that responses from memory can be routed back to the correct core which originates the request.

- **Arbitration Policy:** The arbitration policy implemented is round-robin scheme which is able to provide fair and deadlock-free access to memory resources.

- **Protocol Compliance:** AXI protocol is fully compiled, involving also support for out-of-order responses, burst transactions and handling of AXI channels (AW, W, B, AR, R).

- **Scalability:** The system is easily extended also to support additional cores when adjusting its parameters.

As seen in Figure 4.3 when both cores issue memory requests, the arbiter alternates grant access to Core 0 in first cycle, and then to Core 1 in the next cycle in a cyclic order. If only one core have issued a request in a particular time event, the arbiter is going to grant access to that core immediately without waiting for the other one. By this implementation is achieved a balance between fairness and efficiency, thus is avoided complex prioritization logic while still maintaining responsiveness to active cores.

Each memory request is associated with a unique ID to support proper routing back of memory responses to the originating core. When the arbiter receives a request it forwards it with the help of multiplexer logic to the memory system, which keeps track of the associated Core ID. This tagging mechanism makes possible for the memory responses to correctly match the originating core, maintaining consistency and correctness in core-memory communication.

This methodology provides several advantages. First, it guarantees fairness by preventing any core from dominating access to the shared memory. Second, the simple round-robin algorithm makes it easy to implement and verify, especially in hardware design environments like SystemVerilog. Third, the deterministic behavior of the arbiter simplifies debugging and performance analysis. However, this approach also introduces some trade-offs.

Specifically it may increase memory access latency under some certain conditions. For example when one of the cores is IDLE and the arbiter waits unnecessarily amount of time to grant access. Additionally, some bottlenecks are noticed related with the lack of dynamic prioritization which means that real-time or latency-sensitive tasks might not be optimally served.

Overall, this implementation with arbiter scheme, shared memory access, along with response routing via Core IDs, has provided an accurate and balanced solution for managing memory access in a multi-core CVA6 architecture. It lets the system to maintain stability while enabling both cores to interact with the shared subsystem in an conflict-free manner.

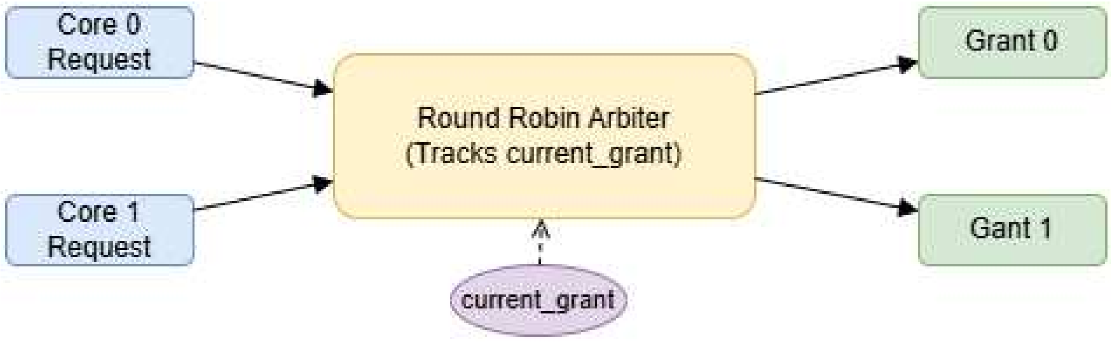


Figure 4.3: Round-Robin Arbitration

4.2.3 AW-Channel-Based Cache Invalidation for Inter-Core Coherence

Maintenance of data coherency in multi-core systems when multiple cores access shared or overlapping memory regions is one of the most critical challenges this architectures faces. This is addressed through, an AXI invalidation implementation which is a component designed to facilitate memory consistency in a multi-core system-on-chip (SoC). Each core may independently issue memory access requests, potentially leading to circumstances where one core's memory operations can invalidate or interfere with the cached data of another core. So to ensure that all cores observe a consistent view of memory is fundamental for correct system operation.

Key Functions

- **Monitoring Memory Accesses:**

The system continuously monitors and tracks memory access requests (specifically, write address requests) from both cores. Each core provides its request address and a valid signal that indicates whether the request is active.

- **Invalidation Signaling:**

As soon as one of the cores issue a valid memory access request, an invalidation signal is generated and the accessed address is forwarded to the other core. As a

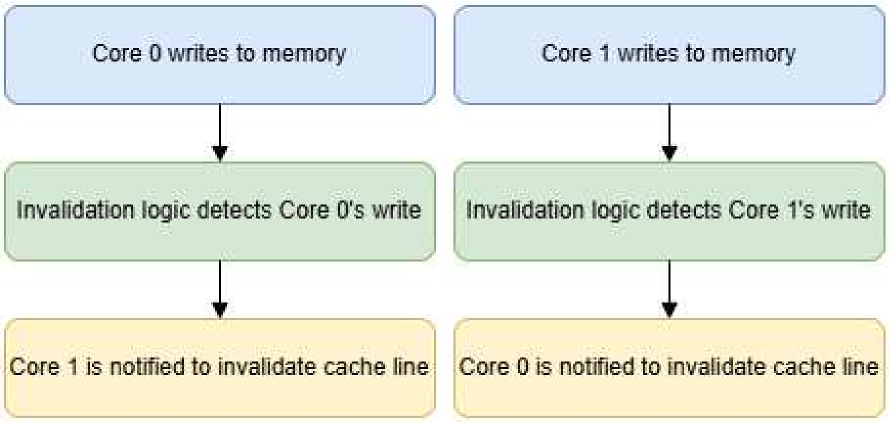

Core 0 writes to memory
Core 1 writes to memory
Invalidation logic detects Core 0's write
Invalidation logic detects Core 1's write
Core 1 is notified to invalidate cache line
Core 0 is notified to invalidate cache line

Figure 4.4: Invalidation Mechanism

result the other core is notified that its cached data corresponding to the given address may be stale and should be invalidated or updated.

- **Selective and Efficient Filtering:**

It is important to make sure that invalidation is raised only for valid and meaningful requests, so this way can reduce unnecessary coherence traffic and improve system efficiency.

Architectural Role Coherence is crucial when dealing with a multi-core CPU that has private L1 caches. At the interface between the memory subsystem and the cores, the module serves as a thin coherence filter. It provides a straightforward but efficient method of preserving communication without the hassle of a full directory-based or spying protocol by sending out invalidation signals based on observed memory accesses.

Benefits of this implementation are:

- **Simplicity:** The design is straightforward, and is suitable for small-scale multi-core systems.

- **Scalability:** While the current implementation covers two cores, the underlying logic can be extended to more cores with some modifications related to the architecture.

- **Performance:** By taking in consideration only relevant invalidation events, the mechanism minimizes unnecessary cache invalidation and coherence traffic, thus improves the overall system performance. This scheme is explained in the flowchart at Figure 4.4.

In summary, this mechanism acts as a critical building block for efficiency and correct cache coherence in a dual-core RISC-V system. It makes sure that each core is properly informed of memory accesses by its peer, thereby maintaining a consistent and up-to-date view of shared memory resources.

4.2.4 Shared Memory Subsystem

The shared memory subsystem incorporates together two primary components first the AXI-to-memory bridge and secondly the parameterized static random-access memory SRAM. The AXI-to-memory bridge is an interface between the AXI bus and the memory. It translates AXI protocol transactions into native memory requests that can be further processed by the SRAM. This bridge is able to support concurrent read/write operations from both cores, and can ensure that memory bandwidth is effectively utilized. Both cores can access memory with minimal contention.

Then the SRAM structure is configurable,and lets that parameters such as memory size, data width, and access latency to be adjusted based on system specifications. At simulation startup, it is initialized using DPI-C (Direct Programming Interface for C) functions, which facilitate the loading of ELF (Executable and Linkable Format) binaries directly into memory. Through this mechanism the simulation of realistic software execution environments can be possible, as actual programs can be loaded and executed by the cores during verification and testing process.

Key features of the memory subsystem include:

- **High Throughput and Low Latency:**

The subsystem provides high-bandwidth, low-latency access for both cores,and minimizes memory access delays.

- **Burst and Single-Word Transaction Support:**

The bridge supports both burst transactions (where multiple data words are transferred in a single transaction) and single-word accesses. This flexibility allows the subsystem to cover a wide range of memory access patterns, starting from sequential program and continuing to random data accesses.

- **Flexible Initialization for Testing and Validation:**

The use of DPI-C for memory initialization results in a flexible mechanism for loading test programs, firmware, or arbitrary data patterns into the SRAM. This feature enables the simulation environment to be easily configured for different test scenarios, including also corner cases and application-level tests.

Overall, this subsystem is designed to be both high-performance and flexible, while supporting all requirements of multi-core systems and providing comprehensive verification through realistic and configurable simulation setups.

4.2.5 Debug Infrastructure

Verification and debugging phases of the design became easier, when utilizing UVM's messaging features, especially UVM_INFO macro which can a give clear view of what's happening in the dual-core memory system. Rather then relying only on analyzing waveforms or manual logs, the system can also automatically produce informative messages whenever important memory events happen. These messages are strategically generated to report on transaction initiation, arbitration outcomes, and invalidation events, ensuring that all important system activities are captured in a manner that is accessible within the UVM reporting framework. This method is systematically using this calls to log metadata for every operation, including transaction IDs, memory addresses, burst lengths, instructions and originating cores. Every information relevant for building and debugging the system such as arbitration, invalidation or inter-core logic is reported via UVM messages.

This integration contributes in connection between high-level functional events and low-level signal activity, especially when they are combined with waveform viewers. This results in an environment where verification, performance analysis, and debug processes are all aligned. This changes made significantly easier identification of common issues related to memory access, arbitration, and coherency in a complex multi-core system.

Benefits:

- **Verification:** UVM messages capture every important memory event under any possible specification or requirement.
- **Performance Analysis:** With this implementation the project can be analyzed deeply including arbitration, memory latency, and system throughput in a range of workload scenarios.
- **Debug and Diagnosis:** UVM reporting system shortens debugging time by making it easy to find/analyze the root cause of bugs and performance bottlenecks.

Overall, this debugging methodology is crucial for the verification strategy because can provide additional correctness, performance tuning, and accelerating the resolution of complex multi-core interaction issues.

4.3 Main Pipeline Stages

The processor's core pipeline is organized into many stages, each of them responsible for specific aspects of overall system performance.

- **Frontend (Speculative Regime):**

  This stage fetches the instructions and prepares them for decoding. Instruction Scan (IS) unit, which fetches instructions and manages execution paths is one of the most significant components of this stage. Branch prediction and speculative control flow are managed by the Branch History Table (BHT), Return Address Stack (RAS), and Branch Target Buffer (BTB). The PC Select and Re-aligner components handle program counter selection and instruction alignment. They make sure having an accurate instruction flow. Then, the CSR Write unit updates Control and Status Registers (CSRs), which mange exception and control the system.

- **Instruction Decode (ID):** Fetched instructions are decoded and processed in this phase. They are positioned in the instruction queue, and are prepared for execution. Compressed Decoder handles compressed RISC-V instruction formats in order to optimize instruction bandwidth. Here instructions are correctly decoded and ready for the next step.

- **Issue Stage:** Now decoded instructions are dispatched to some functional units. The Issue Buffer temporarily stores and issues them in order, with the aim of having a smooth pipeline operation. The Scoreboard monitors their dependencies and provides their dispatching only when all operands are ready. This way is maintained data consistency and is prevented any potential hazard.

- **Execute Stage Pipe:** Functional units (FUs) in this stage perform arithmetic and logical operations. Under this units are included the Arithmetic Logic Unit (ALU), CSR Buffer, Floating Point Unit (FPU), and multiplication/division units. Memory operations are managed by the Load-Store Unit (LSU), which handles load and store instructions to ensure seamless data access.[5]

- **Out-of-Order Writeback (OoO WB):** This stage handles the final step of instruction execution. It writes functional results back to registers or memory, and manages critical operations such as page table walks (PTW) and virtual address translation. [5]

- **Commit Stage:** The commit stage puts instruction execution in program order. It includes robust exception handling mechanisms to manage system events and exceptions, maintaining system stability. Additionally, a privilege check ensures that privileged operations are executed correctly and securely, preserving system consistency and enforcing access control.

- **Cache Subsystem:** The shared memory subsystem spans across all stages. Instruction Stream (IS) and Data Stream (DS) handle cache requests and manage data for instruction fetches and memory operations. The DS Miss Unit and DS Controller are relevant in handling memory access and resolving any possible cache misses. They ensure smooth data flow and reduce latency across the core pipeline.

- **Key Features Highlighted in the Diagram:** The core architecture incorporates several advanced features together to achieve the desired performance metrics. Speculative execution provided by branch prediction mechanisms, increases pipeline efficiency by executing instructions based on predicted control flow.

  Out-of-order write back decouples execution from commit for a better resource utilization. Integrated caching mechanisms support efficient instruction and data access, reducing memory latency. The dedicated logic handles interrupts and exceptions.

This architecture emphasizes a well-optimized, pipelined RISC-V core with support for speculative execution, efficient functional unit utilization, and robust cache and interrupt management.[4]

4.4 UVM Approach Specifications

UVM verification for dual-core RISC-V architecture relies on a modular testbench structure that can be mapped to various parts of the CVA6 core pipeline for comprehensive verification. To develop the environment further for multi-core setup and to incorporate the new requirements, is needed to focus on several areas of the current architecture.

4.4.1 Multi-Core Verification

In order to make this approach convenient is modified the UVM environment to support multiple CVA6 cores(in this case integrating 2 cores) by defining an array of configurations.

Each core has its own specific settings which help them integrate in the multi-core setup. Secondly interface replication is performed, and as a result all necessary interfaces such as clock, reset, debug and AXI are replicated as well for each core. So they have their own set of interfaces for proper communication and control.

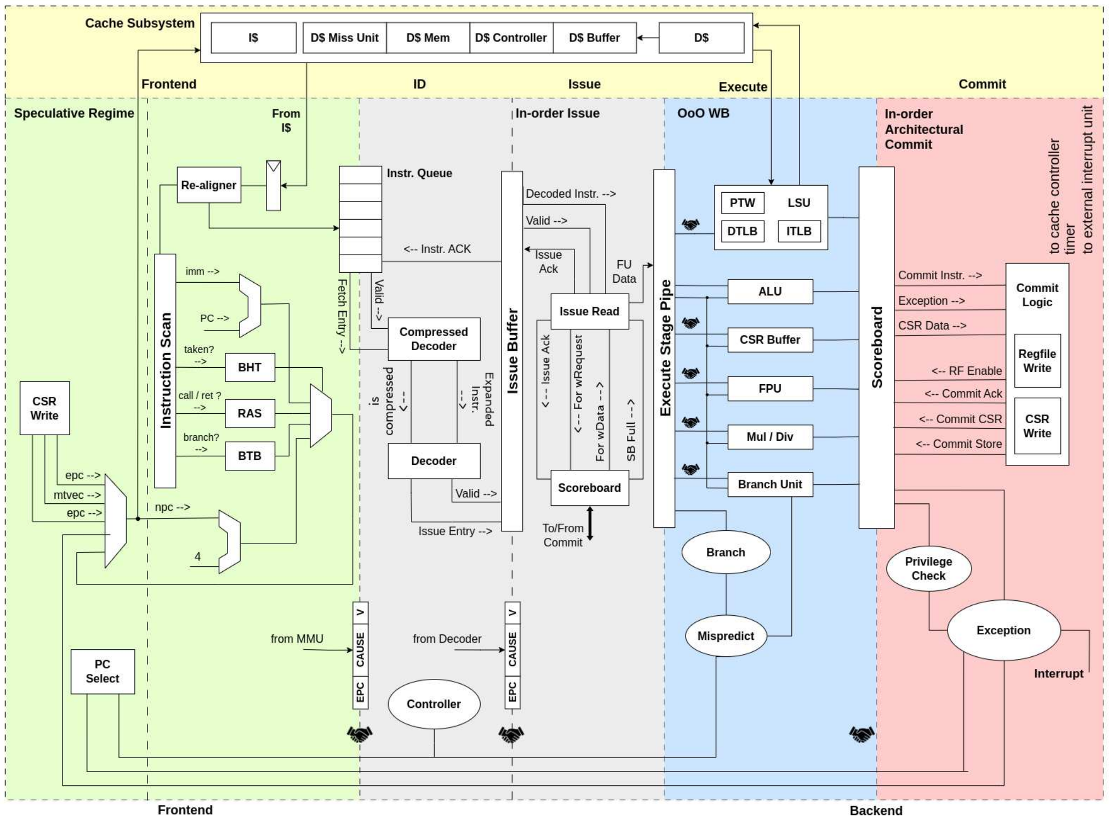


Figure 4.5: CVA6 core overview

The approach focuses on modularity and scalability. Key principles used are:

- **Hierarchical and Modular Design**

Cores are equipped with the same verification block, they have the same agents, and follow the same modular approach that supports the implementation without major changes to the environment. The testbench is like a hierarchical structure, with the test layer at the top, aligned with the environment and core-specific agents, promoting organized and efficient verification.

- **Re-usability**

All relevant agents in this work starting with clkreset, AXI, and interrupt agents are considered as reusable components. If in the future the approach is going to increase the number of cores inside the design the same agent types can be reused. This feature reduces redundancy in the environment.

- **Separate Operation**

Each core operates independently of their peer, but all the results are consolidated in the scoreboard. This setup allows a separate operation and execution in terms of verification across cores.

- **Coverage**

The setup is adapted and organized to cover each core functionality and interaction. And it is able to provide for any operation, structure, module or function the amount of the logic covered by coverage metrics.

- **Shared Resource Verification**

The AXI interface and shared memory subsystem facilitate communication among cores. Agents and the scoreboard collaborate to verify correct arbitration for shared resources, maintaining timing and data integrity for AXI transactions, and detecting and handling potential deadlock and race condition scenarios.

4.5 Challenges

1. **Complexity in Synchronization:** Multi-core systems often need synchronization between components. Verifying such scenarios (e.g., shared memory access contention or interrupt synchronization)requires additional checks. Handling race conditions and deadlocks in this approach is particularly challenging.

2. **Performance Bottlenecks:** Verifying multiple cores simultaneously increases runtime and complexity of simulation. This can lead to performance bottlenecks if the system is not optimized correctly.

3. **Shared Resource Contention:** Verifying shared memory and resource contention scenarios requires precise timing checks to ensure data integrity. Designing sequences to explore all edge cases can be time-consuming.

4. **Debugging Multi-Core Interactions:** Debug process in multi-core verification is challenging because bugs are harder to isolate. Because the problem may be caused from core interactions rather than individual core behavior. Debugging such issues requires detailed communication/interaction monitoring across cores.

5. **Maintaining Coverage Goals:** Achieving a high percentage of coverage for both core-specific functionality and system level interactions can be difficult, because it requires significant effort in defining coverage models and sequences.

6. **AXI Interface Complexity:** The AXI interface has multiple handshake logic and timing requirements to be met. Verifying correct behavior for all possible scenarios (e.g., burst transfers, interleaved transactions) across cores increases complexity.

7. **Scalability Limitations:** This modular approach is scalable, but in case of increasing the number of cores the simulation overhead increases as well. In such scenario is relevant to have an efficient resource allocation and test case optimization.

8. **Arbitration Complexity:** Designing an arbitration system that fairly arbitrates between multiple masters (cores) is non-trivial. But ensuring no starvation, deadlock, or priority inversion needs careful policy selection during verification process.

9. **ID and Response Routing:** The extension of IDs so that responses are always routed to the correct originating core, especially under out-of-order and burst transactions, increases complexity to both the multiplexing logic and the tracking logic.

To overcome these challenges is required a combination of careful and detailed architectural planning, robust parameterization, and thorough verification to ensure a correct, efficient, and scalable multi-core RISC-V system.

4.6 Agent Structure and Components

The diagram described in Figure 4.6 represents the basic structure of a UVM (Universal Verification Methodology) agents used to verify the CVA6 core. This structure is fundamental to any UVM-based verification environment because it shows all the building blocks for driving individual cores in the CVA6 multi-core architecture. As seen also earlier each core in such architectures should have its own dedicated agent. Here's a simple explanation of the components involved and their roles.

4.6.1 Multi-Core Verification with Multiple Agents

In this design, each core is managed by a dedicated verification agent. While the basic structure of the agent remains consistent across all cores, multiple instances are integrated within the environment.

These agents operate independently from their peers, and they handle tasks such as driving their respective cores, applying transactions, monitoring outputs, and collecting coverage data.

4.6.2 Sequencer

- **Purpose:** The sequencer role is to generate and send a series of sequence items (transactions) to the driver. These communication between sequencer and driver represent the input stimuli that will be applied to the Design Under Test (DUT).

- **Details:** The term sequence items involves various operations and components such as memory reads/writes and interrupt signals which are stored in a database. The sequencer communicates with the driver through a Transaction-Level Modeling (TLM) port, enabling the efficient transfer of transaction data and seamless execution of test scenarios.

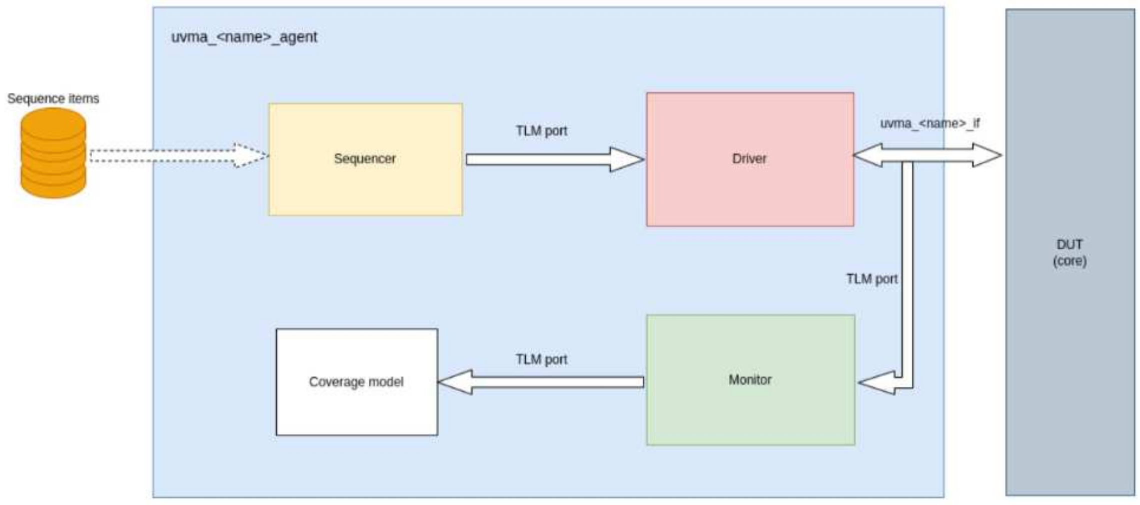


Figure 4.6: CVA6 Agents Overview

4.6.3 Driver

- **Purpose:** The driver serves as a bridge between sequencer and DUT. It receives the sequence items and translates them into real-world signal-level activities on the DUT interface.

- **Details:** The driver interacts with the DUT by ensuring that appropriate control signals or data are applied to the DUT's input ports under correct time constrain. This feature of the driver converts high-level transactions coming from sequencer into low-level signals essential for DUT.

4.6.4 Monitor

- **Purpose:** The monitor passively observes the DUT's outputs and extracts data for subsequent analysis.

- **Details:** It is connected to the DUT's interface and collects information/data during simulation process. It uses a TLM port to send the extracted data to other components such as the coverage model or scoreboard for a detailed analysis. It is a non-intrusive component, that is why the monitor does not drive signals or affects the DUT's behavior. Its aim is just to maintain accurate observation of system operations.

4.6.5 Coverage Model

- **Purpose:** The coverage model is responsible to analyze the data monitored earlier and to measure test completeness.

- **Details:** The coverage model provides a comprehensive verification because it checks whether all DUT components have been exercised and are functionally correct. It keeps track of coverage metrics such as code coverage (e.g., lines of code executed) and functional coverage (e.g., tested scenarios). The outcome from the coverage model acts as a guide towards improvement of test cases by identifying all untested conditions.

4.6.6 Sequence Items

- **Purpose:** Sequence items are able to represent the individual interactions or operations that the sequencer has generated.

- **Details:** It specifies some relevant parameters such as data, addresses, or control signals to be applied to the DUT. They are stored in a database and are acquired by the sequencer when needed.

4.6.7 Interfaces

- **Purpose:** The interface acts as the connection key between all mentioned components such as the driver, the monitor, and the DUT.

- **Details:** It contains all signal definitions regarding inputs and outputs of the DUT, and makes specifications of the communication protocols. It is frequently used by the driver to apply stimuli and by the monitor to observe the DUT's behavior.

All Components Together

1. The sequencer duty is to generate high-level transactions based on the scenario and eventually to transfer this transactions to the driver.
2. The driver receives and translates these transactions into signal-level activity on the DUT with the help of the proper interfaces.
3. The DUT receives and processes this information and accordingly produces the output.
4. The monitor makes a detailed observation of such outputs in order to extract accurate information from them. Which is later send to the verification components of the design.

Key Advantages of Multiple Agents for Multi-Core Design

- **Parallelism:** Cores are independently verified, this separation allows concurrent operation and reduce verification complexity.
- **Modularity:** Since each core is handled by a separate agent, debugging process becomes more straightforward since all issues are isolated to specific cores.
- **Scalability:** Re-usability of agents for any additional core with minimal changes, makes the environment highly scalable.

When modifying the CVA6 verification environment for multi-core support, significant changes were done. The agent structure is replicated in order to be reused for each of them. Instead of having a single agent instance, with this changes we have arrays of agents to support multiple cores (in this case, two).
The key agents we've replicated for each core are:

AXI Agent: Is responsible to handle AXI bus transactions between cores and shared subsystem.

Core Control Agent: Is responsible to manage specific control signals related to each CVA6 core.

RVFI Agent: Is responsible to monitor the RISC-V Formal Interface during verification process, which deals with instructions.

The environment involves a shared agent labeled "clknrst agent", which provides common clock and reset to all cores in order to have a proper synchronization among all agents.
Agents are defined as arrays in the environment classes and this adaption toward multi-core approach enable the verification environment to simultaneously monitor and control CVA6 cores. The re-usability feature of the agents allows concurrent processing of transactions, coverage collection, and checking across cores.

4.6.8 CVA6 Multi-Core CSR Implementation

To support multi-core functionality in the CVA6 environment, was needed an extension of the logic to provide independent Control and Status Register (CSR) interfaces for each core. This was possible to be reached by using two key macros designed to streamline CSR instantiation and configuration.

Each core maintains its own CSR state to operate independently. Registers like "mstatus" and "mcause" are coupled to core-specific behaviors, such as privilege level transitions and exception handling. This individual usage of CSR interfaces per core, provided an accurate state tracking, eliminated interference, and enabled isolated execution. This architecture also supports robust debug and verification by preserving the integrity of each core's control state.

4.6.9 Critical CSRs in Multi-Core

Most critical CSRs requiring per-core implementation:
Machine-Level:

- mstatus: Machine status register
- mcause: Machine cause register
- mtvec: Machine trap vector
- mepc: Machine exception PC

Supervisor-Level:

- sstatus: Supervisor status
- scause: Supervisor cause
- stvec: Supervisor trap vector

Debug/Performance:

- dcsr: Debug control and status
- mcycle: Machine cycle counter
- minstret: Machine instruction counter

Why Not Sharing CSRs?
When CSRs is shared between cores some significant challenges may arise that can affect system integrity and functionality. This violates the RISC-V privileged specification, which requires each core to maintain its own set of control and status registers. This results in issues related to race conditions from concurrent CSR access, leading to unpredictable behavior.

Moreover, it prevents cores to operate independently and limits the system's ability to execute tasks correctly. Under this conditions the debug process becomes nearly impossible due to the entangled states of shared control signals. Moreover the privilege-based security mechanisms is compromised, creating critical vulnerabilities.

To address these issues, the current implementation aim is to provide to each core an independent CSR state, thus enabling effective privilege management and debugging. Another disadvantage with sharing Control and Status Registers (CSRs) across cores is related with difficulties in verifying per-core behavior, undermining the ability to monitor and manage core-specific operations.

Conclusion

Control and Status Registers (CSRs) must remain independent for each core in order to ensure architectural correctness and maintain proper system functionality. Independent CSRs are essential for accurate exception handling, effective core isolation, and robust debugging capabilities.

Additionally, maintaining this separation is a fundamental requirement for RISC-V compliance. Even if only one core sets the CSRs, the state information is inherently core-specific and cannot be shared without violating critical architectural principles and compromising system integrity.

UVM Test Environment	
RVFI	RVFI
CSR IF	CSR IF
Core 0	Core 1
`mstatus_0`	`mstatus_1`
`mcause_0`	`mcause_1`
`mepc_0`	`mepc_1`
...	...
...	...

CVA6	CVA6
Core 0	Core 1
CSRs	CSRs
mstatus	mstatus
mcause	mcause
mepc	mepc
...	...
...	...

Table 4.1: Multi-Core CSR Configuration

Chapter 5

Results and Waveform analysis

Evaluation of functional correctness and multi-core behavior of the dual-core CVA6 system, was assured by conducting a series of simulation experiments under the developed UVM-based verification environment. On this section are presented the results obtained during simulation, with a particular focus on waveform analysis and inter-core behavior. The waveforms provide detailed insight into the interactions between the two CVA6 cores, the arbitration and routing performed by the interconnect mechanism , and the inter-core coherence access write capabilities of the invalidation logic. By examining these results, is demonstrated how the system manages concurrent memory accesses, ensures that each core is promptly notified of writes by the other and maintains correct operation under various test scenarios. The following subsections show representative waveform snapshots and discuss the observed behaviors in the context of the design objectives.

5.1 Waveform analysis

In the waveform below is shown the behavior of the components needed to implement the dual-core approach. It is important to mention that each core has a unique HART ID, making them distinguishable in the test program and active during simulation. The first block of siganls in the waveform belong to arbitration logic. As known in this approach is implemented a round-robin arbitration scheme, and during investigation of the waveform , we can see that at that specified time stamp core 1 is rising a request and the grant to access the shared memory component is given to core 1. Round-Robin ensures that only one request is granted at a time, and the arbitration works in a cyclic order.

Then the second block of signals in Figure 5.1 belongs to invalidation logic, at the defined time stamp core 1 has written to a specific memory address and has raised an invalidation signal to inform core 0 that it is occupying this address. Lastly in the waveform are shown the instructions (such as fetch, decode ,issue, commit) being executed by each of the cores (highlighted in yellow core 0 and blue core 1).

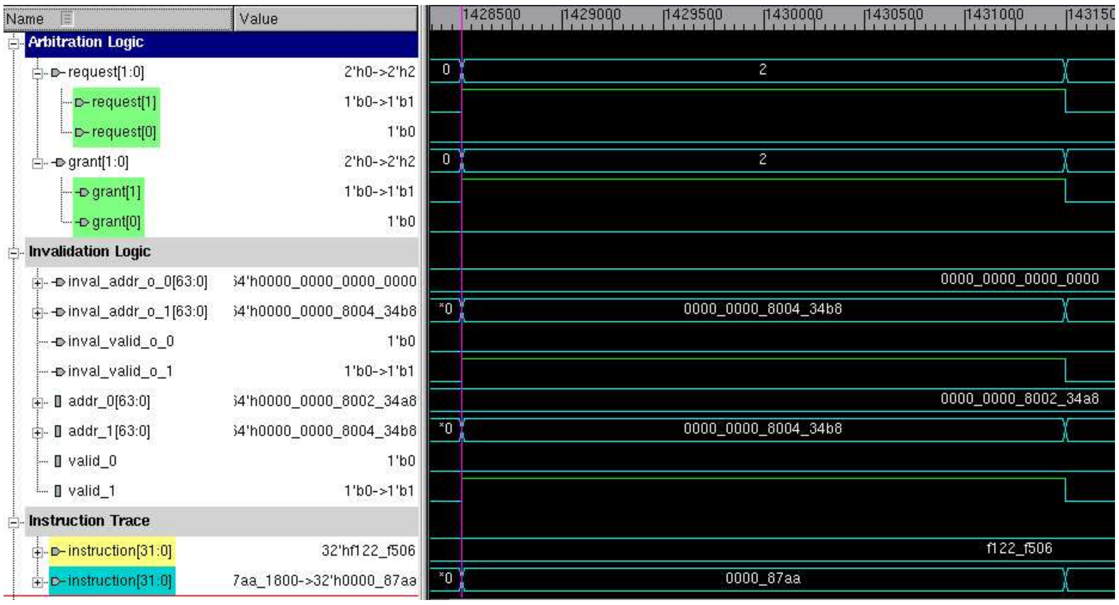


Figure 5.1: Arbitration and Invalidation Logic

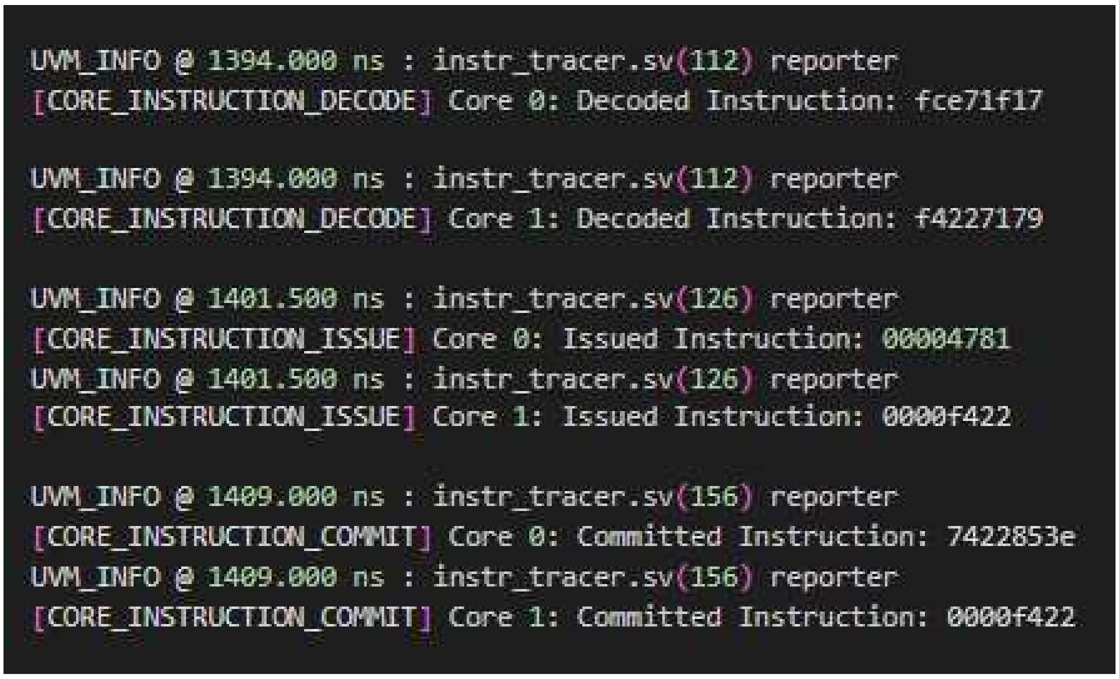
As a result the waveform demonstrate correct arbitration by the multiplexer, alternating between serving Core 0 and Core 1 based on the prepended core ID in the AXI ID field. The invalidation filter successfully detects simultaneous accesses to the memory address. This behavior confirms that the dual-core system, arbitration, and invalidation logic are functioning as intended, with clear visibility into which core is being served at each moment.

5.2 UVM Analysis

These UVM INFO messages in Figure 5.2 are generated by the environment to provide a detailed trace of both CVA6 cores instruction execution in a random time stamp.

These messages collectively show the activity of each core. Specifically demonstrate that in a specific time stamp one of the cores is executing on of the instruction such as decoding,issuing, or committing.

In Figure 5.3 messages are generated by the interconnect mechanism(mux) during simulation to confirm that the AXI ID widths are correctly configured



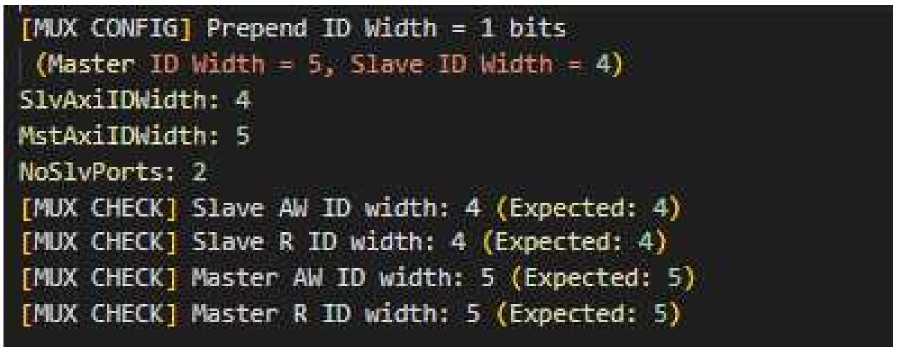
```
UVM_INFO @ 1394.000 ns : instr_tracer.sv(112) reporter
[CORE_INSTRUCTION_DECODE] Core 0: Decoded Instruction: fce71f17

UVM_INFO @ 1394.000 ns : instr_tracer.sv(112) reporter
[CORE_INSTRUCTION_DECODE] Core 1: Decoded Instruction: f4227179

UVM_INFO @ 1401.500 ns : instr_tracer.sv(126) reporter
[CORE_INSTRUCTION_ISSUE] Core 0: Issued Instruction: 00004781
UVM_INFO @ 1401.500 ns : instr_tracer.sv(126) reporter
[CORE_INSTRUCTION_ISSUE] Core 1: Issued Instruction: 0000f422

UVM_INFO @ 1409.000 ns : instr_tracer.sv(156) reporter
[CORE_INSTRUCTION_COMMIT] Core 0: Committed Instruction: 7422853e
UVM_INFO @ 1409.000 ns : instr_tracer.sv(156) reporter
[CORE_INSTRUCTION_COMMIT] Core 1: Committed Instruction: 0000f422
```

Figure 5.2: UVM Infos for Intructions

for both the slave and master interfaces. The first print shows that the MUX is configured to prepend 1 bit to the slave AXI ID (which is 4 bits), resulting in a master AXI ID width of 5 bits. This prepended bit is typically used to encode which core (or slave port) originated the transaction.

Then the slave AXI ID width is 4 bits, the master AXI ID width is 5 bits, and there are 2 slave ports (cores). The MUX CHECK regarding slave port shows that the AXI ID width for the Address Write (AW) and Read (R) channels on the slave side matches the expected value of 4 bits. The MUX CHECK regarding master port shows that the AXI ID width for the Address Write (AW) and Read (R) channels on the master side matches the expected value of 5 bits.

In summary: These checks ensure that the mux is correctly handling AXI ID widths, which is essential for proper transaction routing and response matching in a multi-core system. The prepended ID allows the mux to distinguish which core initiated each transaction when communicating with shared memory or downstream components. And show that both cores integrated in the architecture have their program counter and instruction.

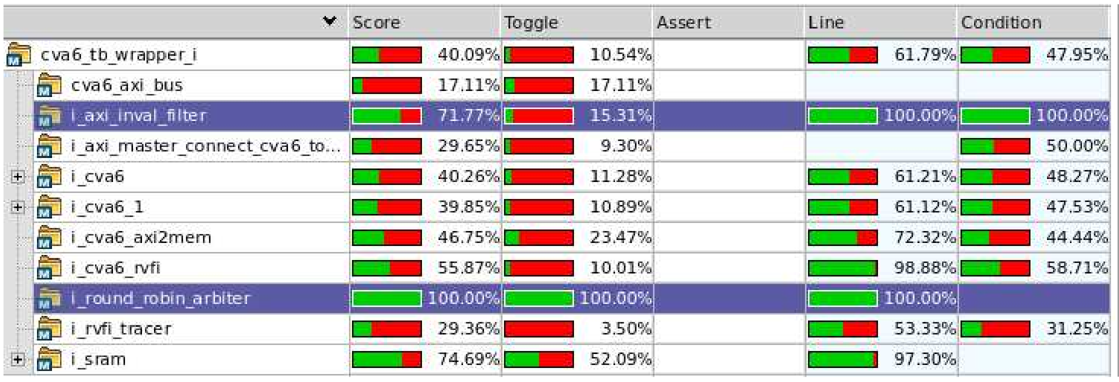


```
[MUX CONFIG] Prepend ID Width = 1 bits
 (Master ID Width = 5, Slave ID Width = 4)
SlvAxiIDWidth: 4
MstAxiIDWidth: 5
NoSlvPorts: 2
[MUX CHECK] Slave AW ID width: 4 (Expected: 4)
[MUX CHECK] Slave R ID width: 4 (Expected: 4)
[MUX CHECK] Master AW ID width: 5 (Expected: 5)
[MUX CHECK] Master R ID width: 5 (Expected: 5)
```

Figure 5.3: MUX configuration

5.3 Coverage Results

To validate the functionality and completeness of the modules being implemented for core-to-memory access control, detailed code coverage analysis was performed using simulation-based verification. As seen in the Figure 5.4 the coverage of both cores implemented in the system is different, because on each of them is running a different test program and each of them have a different Hardware Thread ID to distinguish them on software execution. The modules we are interested to improve the coverage are the highlighted ones, which will be explained below.



	Score	Toggle	Assert	Line	Condition
cva6_tb_wrapper_i	40.09%	10.54%		61.79%	47.95%
cva6_axi_bus	17.11%	17.11%			
i_axi_inval_filter	71.77%	15.31%		100.00%	100.00%
i_axi_master_connect_cva6_to...	29.65%	9.30%			50.00%
i_cva6	40.26%	11.28%		61.21%	48.27%
i_cva6_1	39.85%	10.89%		61.12%	47.53%
i_cva6_axi2mem	46.75%	23.47%		72.32%	44.44%
i_cva6_rvfi	55.87%	10.01%		98.88%	58.71%
i_round_robin_arbiter	100.00%	100.00%		100.00%	
i_rvfi_tracer	29.36%	3.50%		53.33%	31.25%
i_sram	74.69%	52.09%		97.30%	

Figure 5.4: Coverage of the System

Starting with interconnect mechanism which includes arbitration logic and memory response routing. The arbiter achieved maximum line coverage and maximum toggle coverage, both of which have important implications for functional correctness and design robustness.

Line coverage measures whether every line of code was executed during simulation. Achieving maximum line coverage in the round-robin arbiter implies that all logical paths were activated at least once. Both corner and typical arbitration cases were exercised, including:

- Arbitration when both cores request access.
- Arbitration when only one core is active.
- Arbitration when neither core is active.

This indicates that the arbitration state machine and decision logic were fully stimulated under all relevant input scenarios, leaving no dead code or untested branches.

Toggle Coverage Toggle coverage examines whether each bit of every signal switched between 0 and 1 or 1 to 0 during simulation. Maximum toggle coverage for the round-robin arbiter confirms that:

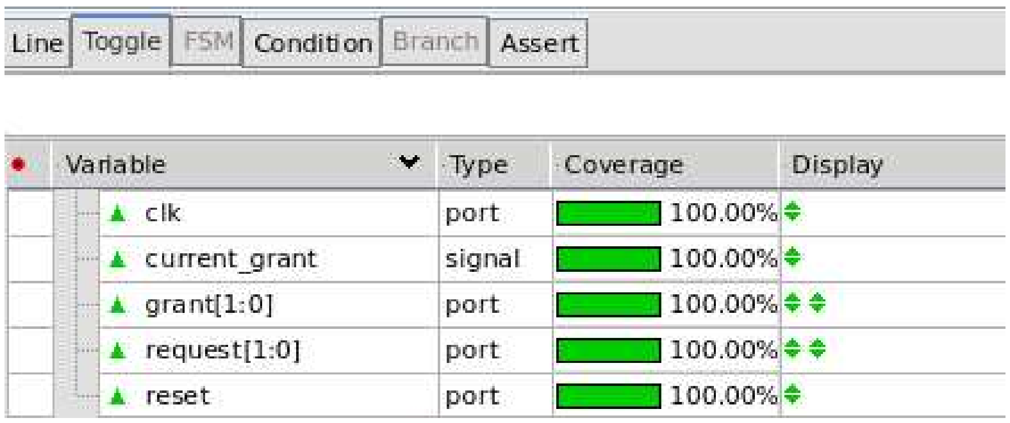
- All internal state variables, grant signals, and control outputs toggled dynamically during simulation.
- Each bit of these signals changed value at least once, indicating active participation in the arbitration cycle.

The round-robin turn logic, typically implemented using a toggling pointer or rotating priority scheme, has fully exercised transitions, such as:

- Alternating access between cores.
- Updating priority pointers or state variables.
- Asserting and de asserting grant lines per arbitration cycle.

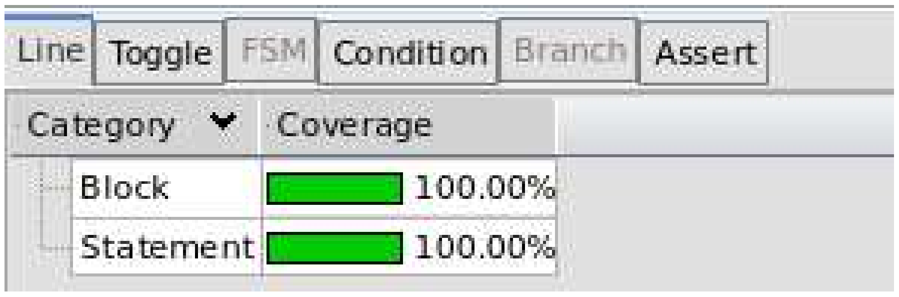
This level of coverage demonstrated on the Figures 5.5 and 5.6 as well shows that the arbiter was tested across all core interaction scenarios, including alternating and repeated requests, and that it handled switching between cores correctly and predictably.
Then moving on invalidation logic the results are as shown on Figures 5.7 , 5.8 , 5.9. Also for this module the coverage includes all three metrics such as line, toggle and condition. Line and condition metric show a maximum result of coverage on both cores whether the condition metric shows a partial coverage on this implementation.

While the line and condition metrics of coverage for invalidation logic reached 100%, indicating that all logic paths and Boolean conditions were exercised during simulation, the toggle coverage remained below 100%. This discrepancy warrants deeper analysis, especially as it reflects the nature of signal activity within the module.



Variable	Type	Coverage	Display
clk	port	🟩 100.00%	
current_grant	signal	🟩 100.00%	
grant[1:0]	port	🟩 100.00%	
request[1:0]	port	🟩 100.00%	
reset	port	🟩 100.00%	

Figure 5.5: Coverage for Arbitration Logic



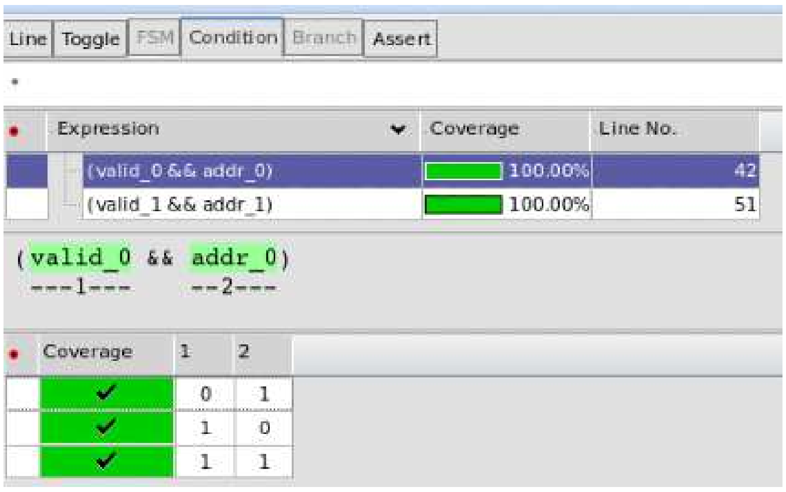
Category	Coverage
Block	🟩 100.00%
Statement	🟩 100.00%

Figure 5.6: Line Coverage for Arbitration Logic

This mechanism was designed to track and forward write access notifications between cores, specifically focusing on:

- Only AW channel signals (such as aw_valid and aw_addr) were used.
- Forwarding address and valid signals to the peer core to trigger invalidation events.
- Other AXI channels — AR (read address), W (write data), R (read data), and B (write response) — were excluded from this implementation, as they were irrelevant to the invalidation logic.
- The address bus width is 64 bits, conforming to the system's AXI specification.

It operates primarily on control signals (e.g., valid, ready), while address signals only reflect activity based on actual memory accesses.The partial toggle coverage arises from the fact that the module monitors only the AXI write address channel, and that only a small subset of the 64-bit address space was used during simulation.

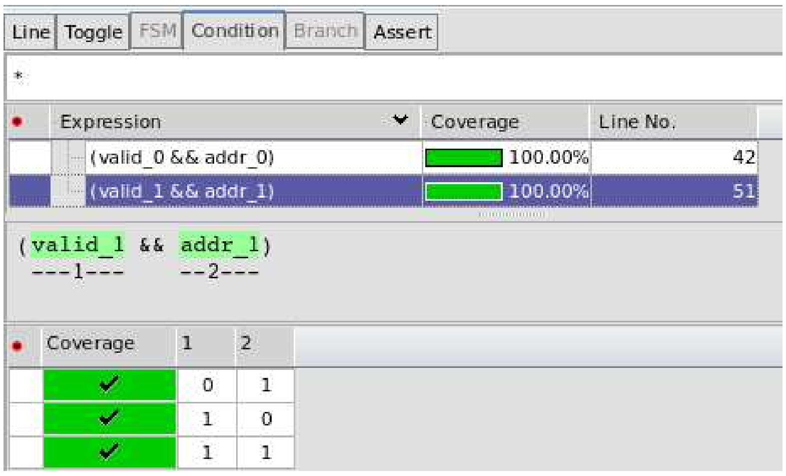


Expression	Coverage	Line No.
(valid_0 && addr_0)	100.00%	42
(valid_1 && addr_1)	100.00%	51

```
(valid_0 && addr_0)
 ---1---    --2---
```

Coverage	1	2
✓	0	1
✓	1	0
✓	1	1

Figure 5.7: Condition Coverage for Invalidation Logic 1st core

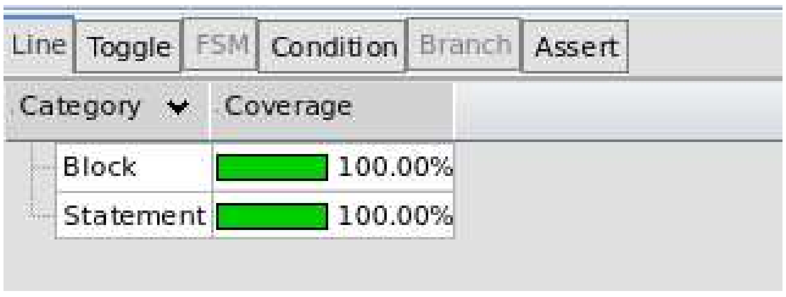


Expression	Coverage	Line No.
(valid_0 && addr_0)	100.00%	42
(valid_1 && addr_1)	100.00%	51

```
(valid_1 && addr_1)
 ---1---   --2---
```

Coverage	1	2
✓	0	1
✓	1	0
✓	1	1

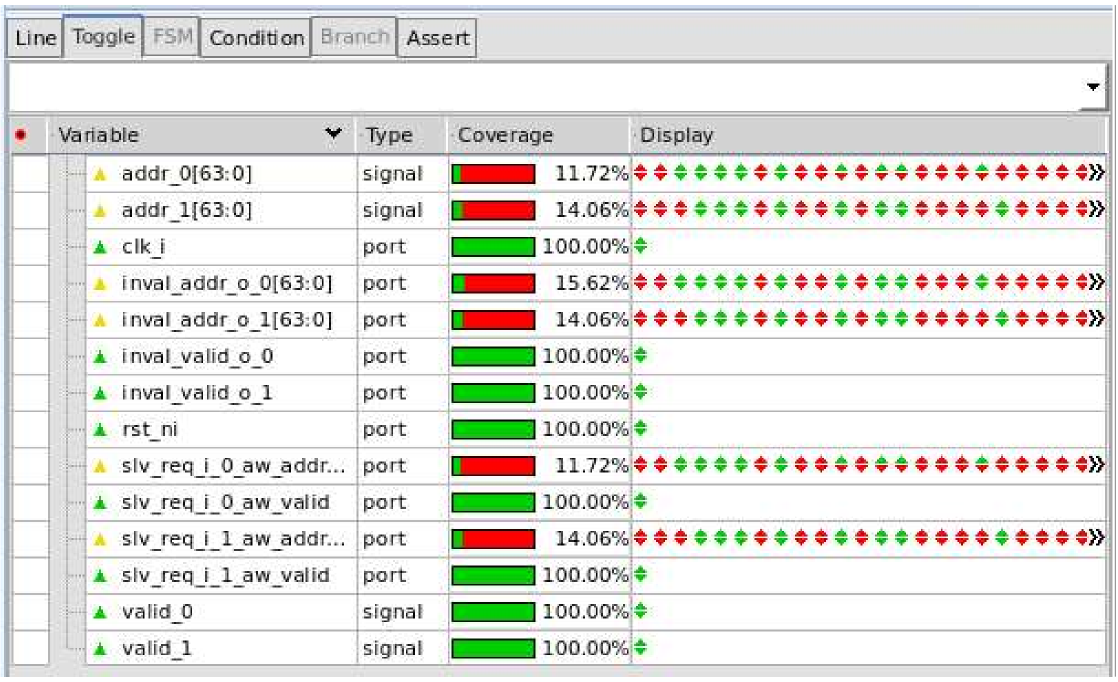
Figure 5.8: Condition Coverage for Invalidation Logic 2nd core



Category	Coverage
Block	100.00%
Statement	100.00%

Figure 5.9: Line Coverage for Invalidation Logic

While the control logic was fully exercised and verified, the full toggle activity of the address lines was not observed due to the limited variation in test inputs. This outcome is consistent with the functional scope and objectives of the current testbench.



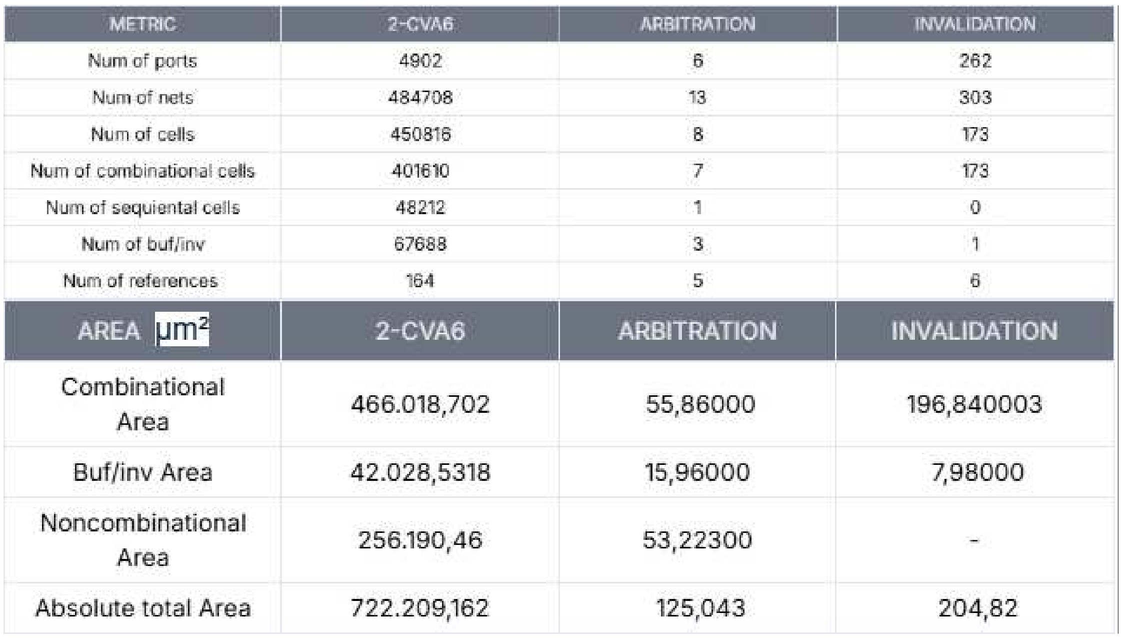
Variable	Type	Coverage	Display
addr_0[63:0]	signal	11.72%	
addr_1[63:0]	signal	14.06%	
clk_i	port	100.00%	
inval_addr_o_0[63:0]	port	15.62%	
inval_addr_o_1[63:0]	port	14.06%	
inval_valid_o_0	port	100.00%	
inval_valid_o_1	port	100.00%	
rst_ni	port	100.00%	
slv_req_i_0_aw_addr...	port	11.72%	
slv_req_i_0_aw_valid	port	100.00%	
slv_req_i_1_aw_addr...	port	14.06%	
slv_req_i_1_aw_valid	port	100.00%	
valid_0	signal	100.00%	
valid_1	signal	100.00%	

Figure 5.10: Toggle Coverage for Invalidation Logic

5.4 Area Report

This section provides a detailed comparison of the area utilization between the dual CVA6 core structure, the arbitration and the invalidation mechanism implemented in the project. The table summarizes the key metrics and areas for all components, expressed in square micrometers (m2). The total combinational area of 2 CVA6 cores is 466,018.702 m2, whereas in the Arbitration, it is significantly smaller, at 55,86 m2. The noncombinational area follows a similar pattern: 2 CVA6 occupies 256,190.46 m2, while the Arbitration mechanism occupies only 53.223 m2.

When the Arbitration mechanism is integrated with the dual CVA6 core system, there is an overall increase in area by 0.018%. This suggests that while it adds functionality, it does a with minimal impact on the overall area. Same result can be mentioned for the invalidation mechanism regarding inter-core coherence in terms of AW (access write) channel of AXI, its combinational are is 196,840003 m2 and it does as well a minimal impact on the overall area even though its functionality is important in terms of what it is needed to be achieved, it increase the area only by 0.03 %.



METRIC	2-CVA6	ARBITRATION	INVALIDATION
Num of ports	4902	6	262
Num of nets	484708	13	303
Num of cells	450816	8	173
Num of combinational cells	401610	7	173
Num of sequiental cells	48212	1	0
Num of buf/inv	67688	3	1
Num of references	164	5	6
AREA um^2	2-CVA6	ARBITRATION	INVALIDATION
Combinational Area	466.018,702	55,86000	196,840003
Buf/inv Area	42.028,5318	15,96000	7,98000
Noncombinational Area	256.190,46	53,22300	-
Absolute total Area	722.209,162	125,043	204,82

Figure 5.11: Area Report

Chapter 6

Conclusion and Future Work

In summary, this work managed to demonstrate the design implementation and verification of a dual-core CVA6 RISC-V system with shared memory access featuring components such as interconnect mechanism for arbitration purpose, and AW-CHANNEL-based cache invalidation mechanism for inter-core coherence. By conducting simulation-based experiments and analyzing detailed waveforms and coverage metrics, the system correctly proved the ability to arbitrate memory requests, detect simultaneous accesses, and maintain correct operation specific workloads. The use of verification methodology UVM combined with RVFI tracing logic provided a complete visibility into core interactions and protocol compliance. Functional and structural code coverage brought a significant help in ensuring and improving verification completeness allowing detection of uncovered and untested logic. However, there is always space for improvement, such as avoiding potential deadlocks in arbitration in case of scaling the environment to 100s of cores. Is valid to be mentioned that in such scenarios will be needed an enhanced cache coherence mechanisms. Future work of this project will focus on scaling the architecture to support and cover more than two cores (ranging from tens to hundreds). Eventually in this implementation of many cores will be necessary to refine the arbitration and invalidation logic to properly handle more complex scenarios, and to implement the integration of complete hardware cache coherence protocol. Additionally, extending the verification environment to include more stress tests and formal verification techniques will further contribute to the improved robustness and reliability of the multi-core system.

Bibliography

[1] *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. RISC-V Foundation. May 2011 (cit. on pp. i, 7).

[2] Jonathan Balkind, Michael McKeown, Yaosheng Fu, and Tri Nguyen. «OpenPiton: An Open Source Manycore Research Framework». In: () (cit. on pp. 2, 3).

[3] RISC-V International. *The RISC-V Instruction Set Manual: Privileged Architecture, Version 1.12*. Section: Memory Model Explanation. Accessed: 2025-06-27. 2021. URL: `https://five-embeddev.com/riscv-user-isa-manual/Priv-v1.12/memory.html#sec:memorymodelexplanation` (cit. on pp. 7, 8).

[4] Florian Zaruba and Luca Benini. *The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology*. IEEE. July 2019 (cit. on pp. 8–10, 16, 25).

[5] J. L Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach (6th Edition)*. 2017 (cit. on p. 24).