FORMAL VERIFICATION OF A MESI-BASED CACHE IMPLEMENTATION

A Thesis

by

VENKATESHWAR KOTTAPALLI

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Aakash Tyagi
Committee Members,	Duncan M. Walker
	Jiang Hu
Head of Department,	Dilma Da Silva

August 2017

Major Subject: Computer Engineering

ABSTRACT

Cache coherency is crucial to multi-core systems with a shared memory programming model. Coherency protocols have been formally verified at the architectural level with relative ease. However, several subtle issues creep into the hardware realization of cache in a multi-processor environment. The assumption, made in the abstract model, that state transitions are atomic, is invalid for the HDL implementation. Each transition is composed of many concurrent multi-core operations. As a result, even with a blocking bus, several transient states come into existence. Most modern processors optimize communication with a split-transaction bus, this results in further transient states and race conditions. Therefore, the design and verification of cache coherency is increasingly complex and challenging.

Simulation techniques are insufficient to ensure memory consistency and the absence of deadlock, livelock, and starvation. At best, it is tediously complex and time consuming to reach confidence in functionality with simulation. Formal methods are ideally suited to identify the numerous race conditions and subtle failures. In this study, we perform formal property verification on the RTL of a multi-core level-1 cache design based on snooping MESI protocol. We demonstrate full-proof verification of the coherence module in Jasper-Gold using complexity reduction techniques through parameterization. We verify that the assumptions needed to constrain inputs of the stand-alone cache coherence module are satisfied as valid assertions in the instantiation environment. We compare results obtained from formal property verification against a state-of-the-art UVM environment. We highlight the benefits of a synergistic collaboration between simulation and formal techniques. We present formal analysis as a generic toolkit with numerous usage models in the digital design process.

DEDICATION

To my parents, my sister, my extended family, and my friends.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to Prof. Aakash Tyagi for his constant support throughout the duration of my graduate study. His persistent optimism and encouragement, prodded me on, especially during periods of disbelief and doubt. This thesis would be improbable without his guidance and help.

I am forever indebted to Prof. Flemming Andersen for providing direction and technical expertise. I am grateful for his friendly, jovial, and immensely supportive mentorship. Prof. Andersen, through weekly interactions, has constantly inspired patience, dedication, and clarity of thought in me. I am grateful to Prof. Michael Quinn for his support, supervision, and encouragement with the UVM simulation effort. I wish to thank Prof. Walker and Prof. Hu for serving as members of my thesis committee, and providing valuable feedback. Additionally, I am thankful to Prof. Sunil Khatri for rousing my creativity and interest in research, while helping me identify my goals.

I am grateful to Surakshith M. Narasegowda, Abhinav Sethi and Sheena Goel, for providing critical assistance at different stages of this project. Surakshith and Sheena helped with the execution of dynamic verification, while Abhinav assisted with design changes. I would also like to thank Yuhao Yang and previous designers of the legacy cache implementation.

I wish to thank the ECE and CSE departments at TAMU for giving me an opportunity to extend my skills and evolve. I am thankful to the CSE graduate advising office, Ms. Karrie Bourquin in particular, for assistance with the logistic requirements. Lastly, I am immensely grateful to my family and friends for their encouragement and belief in my abilities.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professor Aakash Tyagi, Professor Flemming Andersen, and Professor Duncan M. Walker of the Department of Computer Science and Engineering (CSE), and Professor Jiang Hu of the Department of Electrical and Computer Engineering (ECE).

The legacy RTL code and specification, used as a starting point for the thesis, was provided by Prof. Aakash Tyagi, and Prof. Michael Quinn. Surakshith M. Narasegowda assisted in implementation of the dynamic verification scenarios listed in Section 3.2. Abhinav Sethi helped with the bug fix described in Section 5.2.2.3. Sheena Goel supported the analysis of coverage, described in Section 5.1.2.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

Graduate study was partly supported by a scholarship from the ECE department at Texas A&M University.

NOMENCLATURE

HDL	Hardware Description Language
RTL	Register Transfer Level
UVM	Universal Verification Methodology
UVC	UVM Verification Component
FV	Formal Verification
FPV	Formal Property Verification
FEV	Formal Equivalence Verification
MSI	Modified, Shared, and Invalid Protocol
MESI	Modified, Exclusive, Shared and Invalid Protocol
SV	SystemVerilog
SVA	SystemVerilog Assertions
FSM	Finite State Machine
LLC	Last-Level Cache
PLRU	Pseudo-Least Recently Used
ABV	Assertion-based Verification
DV	Data Value invariant
SWMR	Single-Write, Multiple-Read invariant

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Caches are vital components of modern processors. They dramatically improve system performance by reducing the number of accesses to main memory. Most microprocessors have several layers of cache to hide the increasing divide between processor and memory performance. In the current era of multi-core systems, we are presented with two architectural choices: a shared memory model or a distributed model. The distributed model dictates that each core has its own private memory. Different cores exchange information using a message passing interface. The shared memory model, which assumes that all cores access a common, single memory, is more popular of the two options. At any instance of time, multiple cached copies could exist in a shared memory system. Therefore, coherence and consistency are crucial in order to ensure correct functionality of such a system. This chapter establishes formal definitions of coherence, consistency and the MESI protocol.

1.1 Cache Coherence

A typical shared memory system is illustrated in Figure 1.1. Multiple agents are connected through an interconnection network. An agent could be a processor, a direct memory access (DMA) block, or an external device that can write and read from memory. All the agents can perform loads and stores to all physical addresses. Each agent has its own private cache, and the last-level cache (LLC) is shared by all the cores.

Multiple copies of a data block can exist in the system. Intuitively, coherence implies that all the agents see the same, correct value of the datum at a given time. This is essential to ensure compliance with the shared memory model, and ease of debug. Additionally, caches are architecturally invisible, dictating that programmers should not functionally identify the presence or absence of caches.

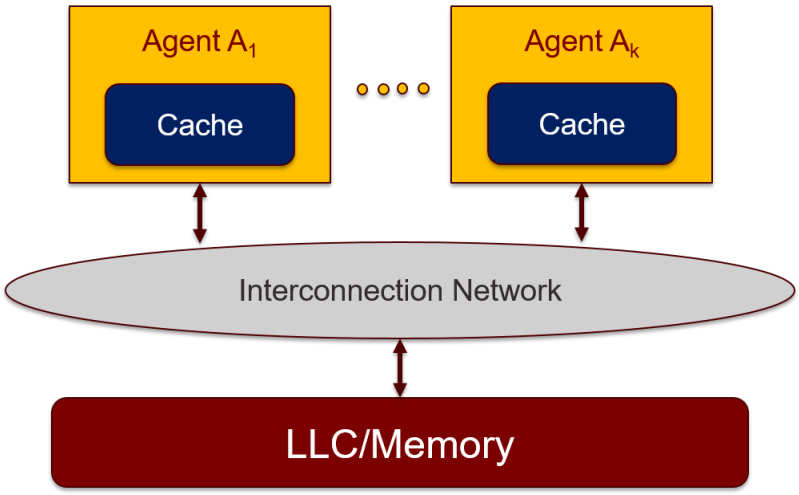

Figure 1.1: A typical shared memory system

Numerous definitions of cache coherence can be found in published literature. But, a coherent system would satisfy all the definitions, as they are equivalent. In this study, our preferred definition of coherence is that offered by Hill et al. [2]. The following invariants compose the formal definition[1] of coherence [2].

1. **Single-Writer, Multiple-Read (SWMR) Invariant:** At a given logical time, a single core can have read-write access or multiple cores can have read only access to a given memory location.

2. **Data-Value (DV) Invariant:** The value of a given data block is the same as the last write access.

Memory consistency, a related concept, is often confused with coherence. Consistency specifies that "a system should appear to execute all threads' loads and stores to all memory locations in a total order that respects the program order of each thread" [2]. A distinguishing feature between coherency and consistency is that coherency is defined on

[1]Invariants SWMR and DV are captured and proven as interface level assertions in our FPV effort

a per-memory location basis, while consistency is specified with respect to all memory locations. Although, it is theoretically possible to have a consistent model with incoherence [3], this topic remains an academic curiosity. Consistency is achieved through coherence protocols. Therefore, cache protocols are vital to the correct design and functionality of a modern processor.

1.2 Coherence Protocols

Coherence protocols enforce the two invariants necessary in a coherent system. Protocols are implemented as a finite state machine(FSM) in each of the storage structures (cache and LLC). A distributed system of cache controllers communicate with each other using transactions and messages to ensure coherence. The type and number of messages and states depends on the particular cache coherence protocol.

In general, there are two classes of protocols, namely snooping-based and directory-based [4]. Snooping-based mechanisms rely on broadcasting messages to all the agents. Most commonly, cores observe the transactions over a shared bus interface and respond accordingly. Collectively, all the controllers ensure coherence. Within directory-based techniques, a central directory holds the state of each data block. State contains information about the current owner (read-write) or sharers (read-only). Requests are point-to-point in directory based controllers. In this study, we consider a snooping-based protocol that relies on a shared bus interconnect, as it is predominant in modern multi-core processors.

Additionally, cache controllers can be classified into invalidate and update protocols, based on action performed in case of a CPU write. An invalidate protocol would request other agents to invalidate their copy of the data block. An update protocol would rather update all existing copies of the block within the system. We restrict our scope to the MESI protocol, which is a popular invalidate technique.

MESI represents the four possible states of a cache block, namely modified, exclusive,

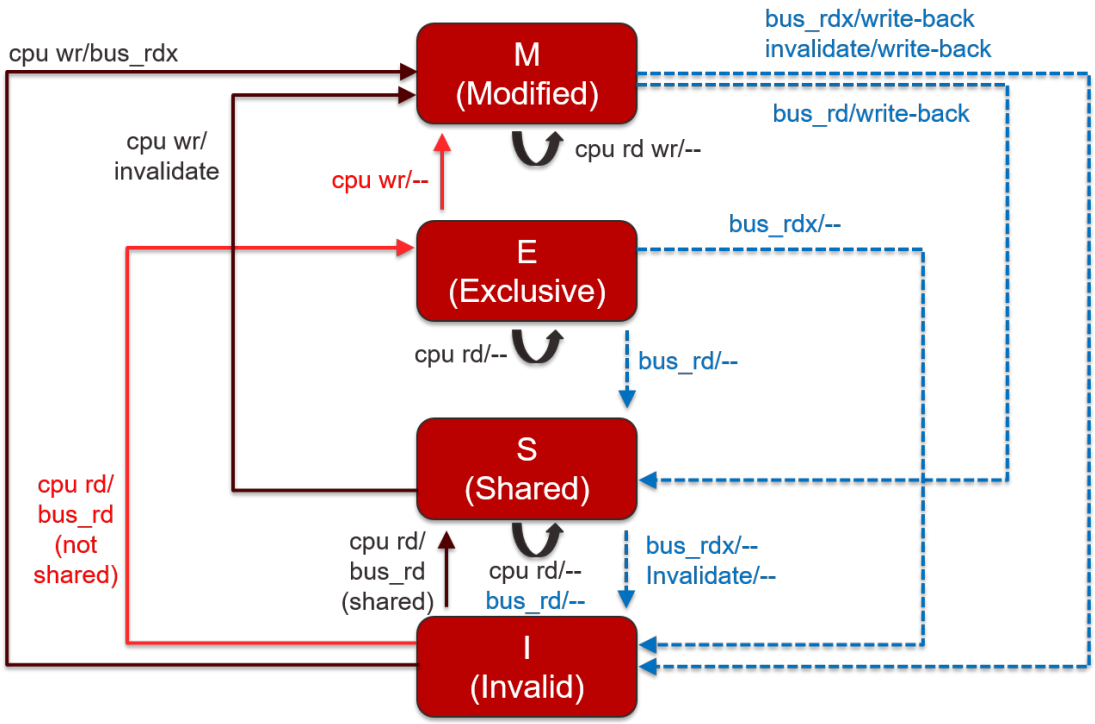

Figure 1.2: The architectural MESI state diagram

shared, and invalid. The architectural state diagram of the MESI protocol is illustrated in Figure 1.2. A processor has read-write access to a block that is in modified or exclusive state in its private cache. At a given time, any one of the agents alone can have a block in modified or exclusive state. A block in shared state provides read-only access to the core.

Transitions to the left of states in Figure 1.2 represent processor side requests and the subsequent controller action on the shared bus. More precisely, read and write are requests from the core; bus rdx (read to modify), bus rd (read-only) and invalidate are requests broadcast on the shared system bus. State transitions to the right on Figure 1.2, signify incoming bus requests and corresponding responses. Collectively, all the cache controllers work together, by adhering to the MESI state diagram, to realize a coherent system.

2. VERIFICATION OF CACHE COHERENCE

In this chapter, we motivate the need for full-proof verification of cache controllers. We present previous work and distinguish it from this thesis in Section 2.2. Thereafter, we describe our approach, objectives, and design implementation in Sections 2.3, 2.4, and 2.5 respectively.

2.1 Need for Robust Verification

The design and verification of cache controllers is notoriously complex [5]. Aside from design and performance considerations, a cache implementation must consider the coherence protocol as well as the communication fabric for functional robustness. The coherence protocol is subject to several race conditions in a truly parallel system with concurrent requests. The MESI protocol defined in Section 1.2 is an abstract model, it simplifies numerous details which are of critical significance in the hardware realization.

Firstly, state transitions are assumed to be atomic in the architectural model. This is impossible to achieve in implementation. The controller would arbitrate for shared resources like the system bus, before performing a state transition. This non-atomicity leads to the existence of transient states. A typical modern processor's cache has about 4 stable states and around 10 transient states [6]. Transient states increase the complexity and lead to additional race conditions. The presence of an atomic bus does not obviate the need for transient states. Although a blocking bus would ensure that not more than one transaction is outstanding on the system bus, state transitions could still be non-atomic. Transient states necessary for a MESI-based controller with an atomic bus is depicted in Figure 2.1 [7]. Transitions due to incoming system bus requests are hidden in Figure 2.1 for simplicity. Race conditions arise when conflicting requests are observed on the processor side and the system bus side. The large number of states and race conditions

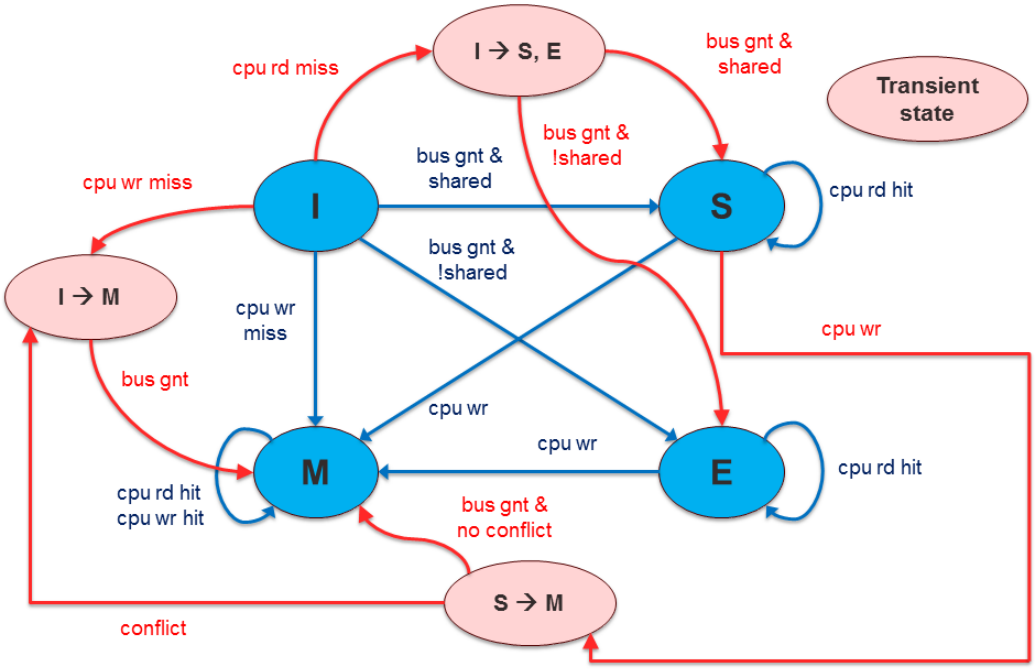

Figure 2.1: Simplified MESI state diagram with transition states

make verification and design extremely challenging.

Second, several performance optimizations are introduced in modern designs. The most prominent example is a split transaction bus. A blocking bus is inefficient from a performance viewpoint. Therefore, modern processors allow multiple outstanding requests on the shared system bus. This would require buffers within the cache controller in order to track outstanding requests. Responses to the system bus requests could arrive out of order. As a result, the design complexity increases tremendously. Recent improvements like write-back buffers, complex interconnection networks, hardware transactional memory, and hierarchical caches further contribute to the design complexity.

Additionally, the communication fabric of the controller is prone to deadlock, livelock and starvation. Therefore, we must also verify the communication infrastructure for

correctness, liveness, and fairness.

Simulation-based techniques alone are insufficient in the verification of caches. They often fail to identify race conditions and critical failures in corner case scenarios. It is challenging to reach confidence in coherence using simulation due to concurrency, and the large number of states and transitions in a cache controller. Vantrease et al. assert that a total of 60 state transitions exist in the simple MSI protocol, when we consider race conditions [6]. Simulation suffers from the lack of controllability, and tedious debug. Simulation does not provide sufficient fine-grained control to exercise interesting corner cases, mainly due to the rigid nature of test-bench components. Certain legal behavior could be prevented by inherent assumptions made within testbench components. We depend on pseudo-random stimulus to fortuitously exercise and identify failing scenarios. The large number of states and transitions makes it improbable to identify all failing corner cases in simulation, leading to critical bug escapes. Even in cases when failures are exercised, debug is tedious and time-consuming due to large traces in which the failure is identified several thousand cycles after the source of error. Coherence design flaws have escaped into shipping products in the past. A popular example is the Intel core2duo coherence bug, "A139: Cache data access request from one core hitting a modified line in the L1 data cache of the other core may cause unpredictable system behavior." [8]. In order to prevent future bug escapes, design teams must ensure robust verification of the complex cache controller.

Formal verification with clever abstraction techniques is often cited as the ideal method for cache verification. There are few documented examples of a verified cache coherence protocol in SystemVerilog or other hardware description languages [5]. Abstract models of protocols have been formally verified in the past with relative ease. However, there is a significant semantic gap between the architectural model and hardware realization. Therefore, we apply formal property verification to guarantee coherence of our MESI-cache

RTL implementation. This was primarily enabled by parameterizing the implementation to solve the complexity problem that would otherwise prevent full-proof verification of the RTL model.

2.2 Previous Work

The design and verification of cache has remained an active research focus for over three decades [9]. Numerous advancements have been proposed with emphasis on either performance or security. Specifically within the purview of cache coherence, several techniques have been explored to make the verification problem more tractable. In this section, we summarize efforts aimed at solving the verification challenge of coherence protocols.

Clarke et al. pioneered the use of formal verification for coherence with the Futurebus+ protocol in 1993 [10]. Thereafter, coherence protocols are typically verified using formal techniques only at the architecture level [11, 12, 13, 14, 15] with tools like Murphi and TLA+. Complexity and the lack of formal tool support for SystemVerilog prohibited effective application of formal methods to the register-transfer-level (RTL) description. At the abstract level, subtle implementation details like timing are excluded to reduce complexity, which in turn facilitates formal techniques. However, there is significant semantic gap between the abstract model and HDL implementation. It is often more challenging to verify coherence of the HDL implementation of a formally verified abstract protocol rather than verifying the architectural model [5]. Once the higher level abstract model is formally verified, equivalence verification is attempted between the RTL implementation and the abstract model. Pong et al. used formal techniques to verify the protocol, but depended on random simulations to verify the implementation [11]. Standard practice is to use simulation for RTL verification of cache controllers. Refinement checkers are developed from the high-level model for use in simulation-based verification of RTL [15]. There are few published examples of formally verified cache hardware descriptions [5]. Dave et

al. suggest the use of high-level synthesis to tackle the verification problem [5]. They demonstrate that automatic synthesis of high-performance, realistic coherence protocols is feasible using Bluespec SystemVerilog (BSV), a guarded atomic hardware description language. Formal verification is performed on a modular, parameterized description in BSV. However, this technique has not been adopted in the industry due to limitations of high-level synthesis.

Recent developments in formal tools, increasing computation power, and main memory size permit a thorough reinvestigation of the applicability of formal techniques. Therefore, we focus on applying clever complexity reduction techniques and standard formal property verification to a parameterized MESI cache RTL design using JasperGold, a model checking tool from Cadence.

In our approach, parameterization enables formal property verification on a reduced configuration of the cache design. Parameterization has previously been used in several studies to simplify the complexity problem and to verify coherence protocols at the abstract level [14, 16, 17]. Safety and liveness properties of several complex cache protocols including split transaction versions have been proven in languages like Murphi. Our approach is unique in the manner that we perform parameterized verification at the RTL level in SystemVerilog, unlike earlier studies which focus on the guarded atomic model. To the best of our knowledge, this is the first study to formally verify MESI-based cache coherence at the RTL level using parameterization.

Alternatively, several efforts have aimed to reduce the complexity and simplify cache coherence. A notable example is atomic coherence by Vantrease et al. [6], which proposes the use of optical mutual exclusion to avoid race conditions completely. Verification is simplified by serializing conflicting coherence requests to the same address. Another example is fractal coherence, which provides a design for verification technique of massively multi-core systems, amenable to formal tools. The DeNovo cache coherence protocol pro-

posed by Komuravelli and Adve [18, 19], places limitations on the allowed parallelism to simplify transient states and non-determinism. Our project is independent of the above techniques. Therefore, our findings can be used to augment and assist validation of the above methods.

Runtime or dynamic verification in actual hardware is also proposed as a means to tackle the complex cache verification problem. Cantin et al. demonstrated the use of additional hardware to detect design errors in addition to manufacturing defects [20]. DeOrio et al. proposed CoSma, a novel technology to provide post-silicon validation of cache coherence protocols in multi-core systems [21]. The drawbacks of post-silicon validation is not limited to hardware and performance overhead alone. Diagnosis of functional errors is challenging due to the limited internal node observability in prototype hardware [22]. Although testing in hardware is orders of magnitude faster than simulation and formal verification, postponing detection of functional errors to post-silicon is restricted by the cost of fabrication.

In 2011, a collaborative initiative between JasperGold and ARM resulted in the formal verification of the ARM AMBA ACE cache coherence [23]. They initially performed abstract protocol verification, and subsequently completed full-proof verification of the HDL implementation. Our study has several similarities to the published version of this effort in many aspects. However, intricate details about the properties and techniques employed by the industry collaboration remain confidential. Therefore, this work contributes by highlighting the properties and methods necessary to prove coherence at the RTL level.

2.3 Approach

The primary objective is to develop a cache controller that adheres to the specification, coherence and a valid communication infrastructure. The interconnect is required to be free of deadlock, livelock and starvation. Additionally, it should obey the dictated input-

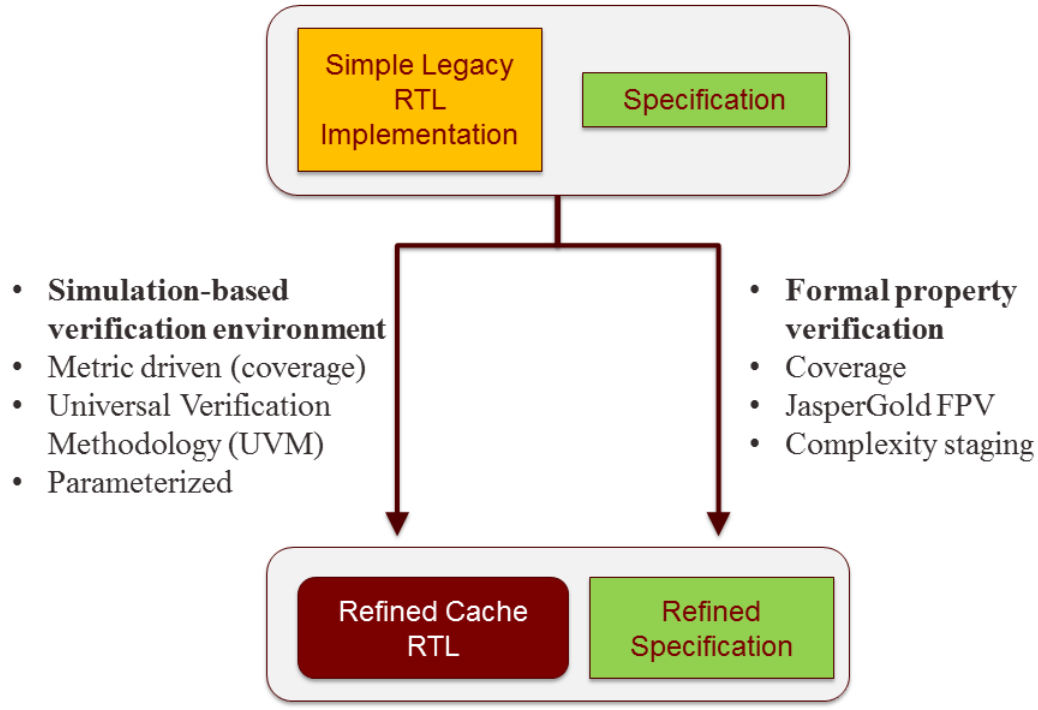

Figure 2.2: Verification approach

output protocol. In this study, we begin with a simple legacy implementation which has been minimally verified with basic test scenarios, similar to how designers in the industry perform a sanity check before release to the verification team. Therefore, the legacy RTL is free from trivial design failures. As illustrated in Figure 2.2, we simultaneously apply simulation and formal techniques to enhance the design as and when we identify bugs. Bugs are essentially deviations from the defined specification. If necessary, we update the specification in case of an architectural flaw. The goal is to complete full-proof verification while eliminating complex corner case bugs and failures.

We offer an effective comparison between simulation and formal methods for the pur-

pose of cache verification at the HDL level. We develop a state-of-the-art, simulation-based environment for this purpose, based on the principles of random, coverage driven verification. Details of the Universal Verification Methodology (UVM) test-bench, a simulation standard predominantly used in the industry, are provided in Chapter 3. The formal verification effort is described in Chapter 4. We apply property verification, with clever abstraction techniques and a sound complexity staging plan, to our design. Parameterization of the design is performed to make formal analysis feasible. Equivalence checking, as reported in Section 4.2 is used to ensure correctness and functionality post-parameterization.

We comment about the return-on-investment of engineering effort in simulation, and formal analysis. Evaluation criteria for the comparison includes, but is not limited to, critical failures identified, ease of debug, length of failing traces, time to develop the environment, and proof of forward progress. Through the effort, we hope to identify methods suited for a symbiotic relationship between the two techniques. We also wish to summarize the design principles required to enable formal analysis of large, complex designs.

2.4 Objectives

The main objectives of this study are outlined below:

- Define properties necessary to guarantee coherence, and correctness of the communication infrastructure
- Identify techniques to achieve a synergistic collaboration between simulation and formal
- Provide a foundation for formal verification of complex, performance-aggressive cache designs
- Highlight design principles necessary to enable large scale formal verification

- Develop an effective comparison between simulation and formal methods for coherence verification at the RTL level

2.5 Design Implementation

The cache implementation under consideration is snooping-based with a shared system bus and uses the MESI coherence protocol. We perform robust verification to transform a readily available, toy design into a synthesizable, bug-free, product-ready design. A block diagram of the envisioned system is presented as Figure 2.3. We strive to develop a robust Register Transfer Level (RTL) description of the level-1 cache and system bus module. For simulation purposes, we use behavioral models of the cores, arbiter, level-2 cache, and memory block. The design is heavily parameterized with numerous options available including number of cores, address width, data width, cache line size, and cache size.

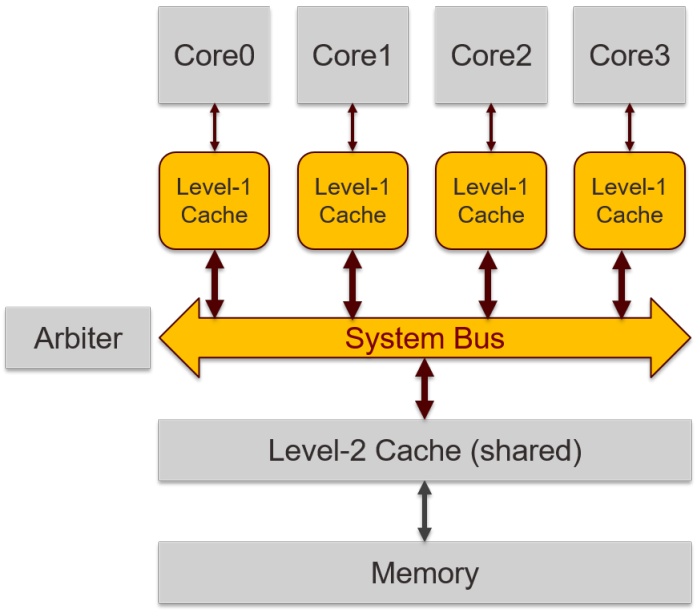

Figure 2.3: System block diagram

Each CPU core is assigned a private level-1 (L1) cache, while all cores share the level-

2 (L2) unified cache. Address space is partitioned into separate instruction and data space. L1 cache is separated into data and instruction cache, but level-2 cache is unified and serves the entire address range. Coherence is maintained through messages on the shared system bus, in adherence to the MESI cache protocol. An arbiter provides exclusive grant of the system bus in a round-robin fashion to ensure fairness. The system bus is atomic; therefore, at any point in time, a maximum of one outstanding request can be pending on the bus. Caches are inclusive, and conform to write-back and write-allocate mechanisms. Additionally, the design policy operates in accordance with a pseudo-LRU (least recently used) cache replacement algorithm. The detailed design specification is included in Appendix A.

3. SIMULATION-BASED VERIFICATION

Simulation is the primary verification technique for integrated circuit design, widely used in the industry for over three decades now [24, 25]. Input stimulus is fed to the design-under-verification (DUV) continuously throughout the duration of the test. Observed outputs are compared with the expected outputs, usually generated by an independent reference model. Considerable amount of infrastructure is required to be developed before any verification can be performed. Common components include stimulus generators, output monitors, checkers, coverage models, and scoreboards. Stimulus generators determine inputs to the DUV at every instance (clock cycle). Output monitors capture the observed outputs into high-level transactions, which are eventually forwarded to the scoreboard. Checkers are embedded within the monitors and scoreboards to detect forbidden behavior or specific failures. Coverage monitors are required to ensure that the entire range of functionality has been verified. Scoreboards generally house high-level checkers and reference models. Most verification components are interdependent. Therefore, substantial time and effort is required even before basic, typical behavior can be verified.

Over the years, substantial improvements in methodology have resulted in increased efficiency and shorter verification cycles. The key technologies responsible are pseudo-random stimulus, intellectual property (IP) based design [26], metric-driven approach, assertion-based verification [27], and standardized methodologies. Improvements are primarily focused on enhancing re-use, predictability, ease of use, and automation. Though often incomplete, pseudo-random stimulus is effective in detecting corner cases and covering a wide range of behavior in a single test scenario. Assertions are critical to detect flaws early in the design cycle. They pin-point to the actual source of the failure, unlike high-level checkers, which usually detect failures late and fail to indicate the source. Lay-

ered stimulus is essential for easy debug and understanding. Standard, open methodologies like OVM, VMM and UVM provide crucial interoperability between tools, companies and vendors. We use present-day industry standard techniques throughout the project to get an accurate comparison between formal and simulation, with respect to cache verification.

In this chapter, we describe the simulation-based verification effort in detail. We develop a state-of-the-art Universal Verification Methodology (UVM) test-bench (TB) built on SystemVerilog (SV). Proven standard industry practices like coverage (metric) driven closure, pseudo-random stimulus, and embedded assertions are rigorously enforced and followed [24, 28, 29].

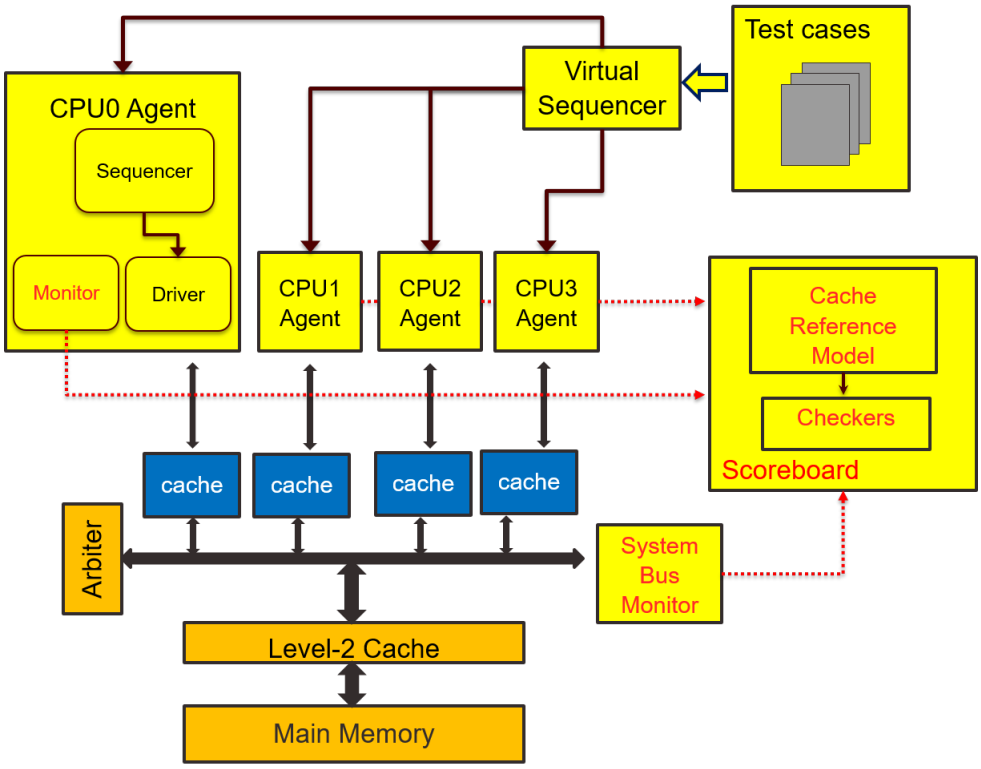

Figure 3.1: UVM verification environment

3.1 UVM Verification Environment

UVM is the industry standard technique used for functional hardware verification [24, 29, 30]. We develop a complete UVM verification environment with pseudo-random stimulus and a metric-driven approach to verify our cache design. A key component of this environment is the transaction-accurate reference model which is used as the golden version. Figure 3.1 represents the overall structure of the test-bench. UVM components developed include CPU agent, virtual sequencer, test cases, scoreboard and the system bus monitor. Behavioral models are used to describe the level-2 cache, memory, and arbiter. The RTL blocks and behavioral models are borrowed from a previous project, while UVM components are devised from scratch. It is important to note that the design (RTL) was partially verified using a testbench-based verification environment in an earlier project with directed testing. This was similar to a minimalistic test-bench typically used by designers to verify basic functionality. Therefore, trivial bugs have been weeded out of the design well in advance. Behavioral models are assumed to be error-free. Hence, the main focus of verification is the multi-core level-1 cache.

The UVM verification components (UVCs) used in the environment are described below.

- **CPU Agent:**

  The CPU agent extends from the 'uvm_agent' base class. It consists of a monitor, a driver, and a sequencer. It mimics the role of a CPU core connected to the level-1 cache in the final SoC design. Sub-components and their functions are listed below:

  Driver: extends from 'uvm_driver' base class. It has access to the CPU-Lv1 interface, through which it interacts with the DUV. It drives signals on the interface based on transactions received from the sequencer. It connects to the sequencer through a handshake mechanism inbuilt in UVM.

Sequencer: extends from 'uvm_sequencer' base class. It receives transactions from the virtual sequencer and forwards it to the driver. It behaves in a sequential manner. Therefore, a maximum of one transaction is driven at any given point of time.

Monitor: extends from 'uvm_monitor' base class. It also has access to the CPU-Lv1 interface, but it can only passively observe the signals. It does not have the capability to affect any of the interface signals. It packages observed behavior into meaningful CPU monitor packets, which are passed onto the scoreboard for high-level checks. Simple interface level checks are implemented as assertions in the monitor. Coverage collectors are also realized in this component.

- **System Bus Monitor:**

  The system bus monitor (SBUS) is a passive component extending from base class 'uvm_monitor'. It observes signals on the system bus interface to create an SBUS packet. This packet is sent to the scoreboard for comparison with the expected SBUS packet. SBUS contains checkers for the DUV outputs on the system bus interface. Collectors are present to monitor coverage on bus activity.

- **Scoreboard:**

  The scoreboard, which extends from 'uvm_scoreboard', houses the cache reference model and high level checkers. It is mainly responsible for ensuring transaction accuracy of the DUV outputs. Two broad classes of DUV outputs are the data returned to the CPU agent, and system bus activity necessary for coherence. The cache reference model mimics the complete functionality of the DUV. Behavioral modeling combined with associative arrays facilitate an efficient implementation in SystemVerilog. The reference model is similar to a guarded atomic model. It assumes that transactions are atomic, which is not always true in the RTL implementation.

It provides the expected data and system bus activity for comparison with observed output packets.

- **Virtual Sequencer:**

  This component is crucial to have fine-grained control of the transactions on each of the CPU agents. It is also referred to as multi-channel sequencer. We define temporal relations between operations on agents through the virtual sequencer. For instance, we can define parallel transactions on each of the CPU agents. It receives a sequence from the test class and forwards transactions to the agents as specified in the virtual sequence.

UVM data items enable communication between the various verification infrastructure components. Three types of transactions are used, namely CPU driver transaction, CPU monitor packet, and SBUS packet. CPU driver transaction is created in the test class and is consumed by the CPU driver. CPU monitor packet, as is obvious from the name, is born in the CPU monitor and consumed by the scoreboard. Similarly, SBUS packet is transferred from SBUS to the scoreboard. Member elements and their corresponding descriptions are presented in Tables 3.1, 3.2, and 3.3.

Table 3.1: Elements of CPU driver transaction

Member	**Description**
Request type	Read or write request
Data	Write data for a write request
Address	Address to be accessed
Cache type	Instruction or data cache access
Wait cycles	Number of clock cycles to wait before driving the transaction

Table 3.2: Elements of CPU monitor packet

Member	**Description**
Request type	Read or write request
Data	Data from cache for read or CPU for write
Address	Address to be accessed
Cache type	Instruction or data cache access
Service time	Number of clock cycles for the entire transaction
Illegal	Flag to indicate if transaction is invalid i.e. write to I-cache

Table 3.3: Elements of SBUS packet

Member	**Description**
Bus request type	BusRD or BusRDX or Invalidate or IcacheRD
Cache number	ID of the primary cache which obtained bus access
Address	Primary address of the bus request
Read data	Data returned to the primary bus request
Request serviced by	ID of the cache which serviced the primary bus request
Copy in cache	Flag to indicate if 'cp_in_cache' was asserted
Shared	Flag to indicate if 'shared' signal was high
Snoop requests	ID of all the caches which requested snoop access
Snoop write back	Flag to indicate if snoop cache performed write-back
Write data snoop	Data of the snoop write-back
Dirty evict flag	Indication if the primary cache evicted a dirty block
Dirty evict address	Address of the evicted cache line
Dirty evict data	Data of the eviction
Service time	Total number of clock cycles taken by the bus transaction

The verification flow for a single CPU operation in this test-bench is depicted in Figure 3.2. The CPU transaction is first received by the driver, which stimulates the DUV input interface accordingly. If the operation requires a system bus message, DUV would request for bus access and initiate the creation of a SBUS packet. Once the bus request is complete, the system bus monitor forwards the packet to the scoreboard. Depending on the primary cache ID, scoreboard pushes the packet into an appropriate queue. Next,

DUV can respond to the CPU agent with the required data(read request) or 'acknowledge' signal (write request), completing the CPU operation. As soon as the transaction is completed, the CPU monitor transmits the monitor packet to the scoreboard. Scoreboard feeds the CPU packet to the reference model, which in turn provides the ideal data and system bus activity. Finally, expected SBUS packet is compared with the observed bus activity by popping an SBUS packet from the appropriate queue.

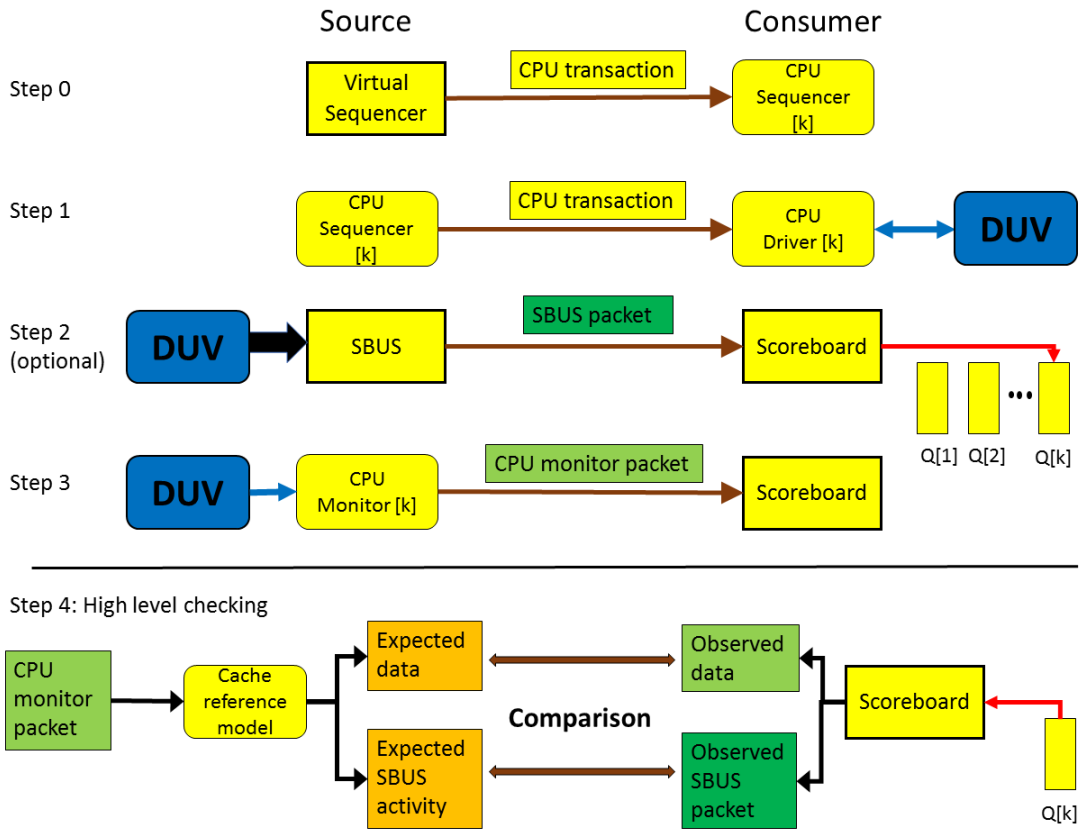

Figure 3.2: Verification flow for a single CPU transaction

A key feature of the verification environment, like the RTL design, is that it is highly parameterized. We can easily configure features like address width, data width, cache size,

length of a cache block, and number of cores. Parameters can be adjusted with no changes to the test-bench infrastructure. Principles of modularity, re-use, and interoperability are followed rigorously by adhering to UVM guidelines.

3.2 Verification Scenarios

A prudent verification plan is crucial for success in the first attempt. In the previous section, we discussed the test-bench infrastructure and verification flow. In this section, we describe critical features, verification methods and specific test cases identified during the planning stage.

Table 3.4: Functions to be verified

Function	**Description**
Basic connectivity	Ensure that the input-output interface is functional.
IO protocol adherence	Design should adhere to the input-output protocol as specified
Read to I-cache	Read operation to the instruction cache
Write to I-cache	Write operation to instruction cache should be ignored
Read to D-cache	Data cache must supply appropriate data to the CPU
Write to D-cache	Data should be written to the correct address
PLRU I-cache	Pseudo LRU replacement policy must be followed by the instruction cache
PLRU D-cache	Pseudo LRU replacement policy must be accurate in the data cache operation
MESI protocol	Cache should respond/send messages on the system bus as necessitated by the MESI protocol (data cache only)

Detailed design specification, provided in Appendix A, is treated as the golden reference. The level-1 multi-core cache design is the main focus of our verification effort. We assume that the level-2 cache (LLC), the arbiter and memory are bug-free behavioral descriptions. As mentioned earlier, both the design and verification environment are parameterized for a flexible number of cores. We concentrate on a 4-core RTL block comprising

of four uni-core cache instantiations. We do not sub-divide the process into different levels of verification. We verify the complete RTL block as one single entity. However, particular test cases would focus only on the functionality of a uni-core cache. We adopt a Grey-box approach in this endeavor, by tapping into few signals internal to the multi-core block. Stimulus is pseudo-random, as scenarios are predominantly pseudo-random with a few directed test cases.

Features identified as crucial for verification are listed in Table 3.4. One or more features are verified in each of the test scenarios described in Table 3.5. Amongst the features, MESI operation is most challenging to verify and debug, due to the sheer number of possibilities. A total of 22 test scenarios were planned and executed. The primary cache and the snooping caches involved were randomized for every test.

Table 3.5: Test scenarios with description

No.	**Scenario**	**Description**
1	Read miss I-cache	- Initiate a read request to a block not present in the instruction cache
2	Read hit I-cache	- Initiate a read request to an instruction block - Wait for the data to arrive in cache - Initiate a read request to the same block
3	Write miss I-cache	- Initiate a write request to a block not present in the instruction cache
4	Write hit I-cache	- Initiate a read request to an instruction block - Wait for the data to arrive in cache - Initiate a write request to the same block
5	Read miss D-cache serviced by L2	- Initiate a read request to a block not present in the data cache of any processor

Table 3.5: Continued

No.	Scenario	Description
6	Read miss D-cache serviced by another core's cache	- Read request to a block on the secondary CPU-1 Case 1: Snoop cache is in M * Write request to block on secondary CPU-1 Case 2: Snoop cache is in E * Do nothing Case 3: Snoop cache is in S * Read request to block on secondary CPU-2 - Read request on primary CPU to the same block
7	Read hit D-cache	-Read request to a data block on primary CPU Case 1: Primary cache is in M * Write request to this block on primary CPU Case 2: Primary cache is in E * Do nothing; block already in E state Case 3: Primary cache is in S * Read request to this block on secondary CPU - Read request on primary Proc to the same block
8	Write miss D-cache	- Select a case randomly Case 1: Secondary CPU cache is in I * Do nothing Case 2: Secondary CPU cache is in M * Write request to the block on secondary CPU Case 3: Secondary CPU cache is in E * Read request to the block on secondary CPU Case 4: Secondary CPU cache is in S * Read request on primary and secondary CPUs * Evict the block from primary cache - Write request to block on the primary CPU
9	Write hit D-cache	- Select a case randomly Case 1: primary CPU cache is in M * Write request to the data block Case 2: primary CPU cache is in S * Read request on both primary and secondary CPU Case 3: primary CPU cache is in E * Read request to the data block - Write request on primary CPU to the same block.
10	LRU read I-cache	- Initiate read requests R1-R8 to 8 instruction blocks of the same set index, ensure that at least 5 of these blocks are unique

Table 3.5: Continued

No.	Scenario	Description
11	LRU read D-cache	- Initiate read requests R1-R8 to 8 data blocks of the same set index, ensure that at least 5 of these blocks are unique
12	LRU write D-cache	- Initiate read/write operations A1-A8 to 8 data blocks of the same set index, ensure that at least 5 of these blocks are unique and that the last 4 accesses are write operations
13	Snoop BusRd request	- Randomly select a case Case 1: Primary CPU cache is in I * Do nothing Case 2: Primary CPU cache is in E * Read request to the block of interest on primary Case 3: Primary CPU cache is in M * Write request to the block of interest on primary Case 4: Primary CPU cache is in S * Read request to the block on secondary CPU * Read request to the block on primary CPU * Evict the block of interest on secondary CPU - Read request on the secondary CPU for the block - Confirm the state transition in primary cache
14	Snoop BusRdx request	- Randomly select a case Case 1: Primary CPU cache is in I * Do nothing Case 2: Primary CPU cache is in E * Read request to the block of interest on primary Case 3: Primary CPU cache is in M * Write request to the block of interest on primary Case 4: Primary CPU cache is in S * Read request to the block on secondary CPU * Read request to the block on primary CPU * Evict the block of interest on secondary CPU - Write request on the secondary CPU for the block - Confirm the state transition in primary cache

Table 3.5: Continued

No.	Scenario	Description
15	Snoop Invalidate request	- Read request to block on the secondary CPU - Read request to block on the primary CPU - Randomly select a case Case 1: Primary CPU cache does not have the block * Evict the block from primary cache Case 2: Primary CPU cache is in S * Do nothing; already in S - Write request on the secondary CPU - Confirm the state transition in primary cache
16	Simultaneous read	- Read request on all CPUs for the same address concurrently
17	Simultaneous write	- Write request on all CPUs for the same address concurrently
18	Round robin write	Generate following sequence to same data block: - Read 0, Read 1, Read 2, Read 3 - Write 0, Read 1, Write 2, Read 3 - Write 1, Read 0, Write 3, Read 2 - Write 0, Write 1, Write 2, Write 3 - Read 0, Read 1, Read 2, Read 3
19	Random single set	- 100 random requests to the same set on all CPUs - Randomize between read and write requests
20	Random test	- 100 random requests on all CPUs - Randomize between read and write requests - Restrict access to two random sets
21	Random delay test	- 100 random requests on all CPUs - Randomize between read and write requests - Restrict access to two random sets - Randomize delay between requests
22	Random six address	- 100 random requests on all CPUs - Randomize between read and write requests - Restrict access to 6 addresses within the same set - Randomize delay between requests

3.3 Checkers

Two broad classes of checkers are employed in this project, namely SystemVerilog assertions (SVA) and transaction-level checks. In general, assertions could either be immediate or concurrent [27]. Since concurrent SVA is suitable for both static and dynamic verification [27], we commit to concurrent assertions for communication protocol checks. The input-output interface verification depends on accurate properties. These assertions are housed within the CPU monitor, system bus monitor, and interface files. The most significant advantage of assertions is that failures are close to the source of error. It is easy to narrow-down the root cause of the bug. However, a disadvantage of SVA is that scope and expressiveness are limited. Therefore, it is challenging to define high-level checks as assertions without additional SV code.

The simulation effort in our project preceded the formal endeavor. As a result, assertions used in initial stages of simulation are used as the starting point for formal property verification (FPV) described in Chapter 4. Nonetheless, several assertions and assumptions were defined on-the-fly during the FPV process. These additional properties were added to the simulation effort gradually. All the assertions used in simulation are indicated in Appendix B. It is important to note that while properties assist in simulation debug, they can significantly increase the run-time of test-cases.

High-level checkers, included in the scoreboard, verify transaction level accuracy in comparison with the cache reference model. DUV outputs can be categorized into 1) data returned to the CPU and 2) messages relayed on the system bus for coherence. We incorporate two checkers, one for each class of DUV outputs. The typical flow for scoreboarding is indicated in Figure 3.2. High-level checks are triggered by the arrival of a CPU monitor packet in the scoreboard. Expected data is compared with the observed data in the CPU packet for a read operation. Expected system bus activity is compared with

the observed SBUS packet. High-level checks engulf a major portion of the functionality. They are capable of verifying a wide range of behavior. However, high-level symptoms are observed several thousand cycles after the root cause of the failure. As a result, it is challenging to debug with merely transaction-level checking.

In summary, we verify our implementation with the below mentioned checkers:

SystemVerilog assertions: Concurrent assertions capture essential properties of the communication protocol interface.

Data integrity check: Data is compared in the scoreboard against the reference model.

System bus activity check: Messages on the common shared bus are crucial for coherence. Activity on the bus is compared against the expected transactions determined by the reference model.

3.4 Coverage Goals

Verification completeness is accessed by a combination of factors like coverage, bug discovery rate, and test-case pass percentage. Verification closure also depends on several business parameters like time-to-market, IC application, risk involved, cost of re-spin, etc. Generally, it is easy to identify additional subtle design discrepancies with extra effort and time. Therefore, it is crucial that the owner/manager decides the right trade-off point between engineering effort and return-on-investment (ROI). Coverage is the most crucial indicator of verification progress [24, 25, 29]. We must define coverage goals during the planning stage, to prevent critical design failures in silicon. However, occasional compromises are made with thorough discussions between all parties involved.

The fundamental principle of a metric-driven approach is to monitor well defined metrics like coverage, pass percentage and bug rate through the verification process. Coverage is broadly classified into code coverage, and functional coverage. The tool implicitly cap-

tures code coverage information from the regression results and RTL. Functional coverage, on the other hand, is explicitly defined by the verification team.

	Code Coverage		
Functional Coverage		Low	High
	Low	Start of the project?	Is design complete? Try formal tools
	High	Add functional coverage: include corner cases	Good coverage: check bug rate

Figure 3.3: Coverage as an indication of verification completeness

Figure 3.3 illustrates progress as a function of code and functional coverage. If both coverage metrics are low, the project is mostly in the early stages with several weeks of work remaining. A high functional coverage coupled with low code coverage indicates the possibility of either dead RTL code and/or poor functional coverage definition. Significant portion of the RTL can be rendered irrelevant if a particular feature is disabled in a legacy IP. In such a scenario, the team must take steps to waive dead code and improve coverage definition. A situation with low functional, but high code coverage indicates that stimulus and behavior is limited. Another possibility is that the design is incomplete. If efforts do not yield any improvement in metrics, it would be wise to try formal tools to eliminate unreachable coverage. The ideal case is one with high code and sufficient functional

coverage. In such a case, the team can call verification closure in consultation with other metrics like bug rate and test-case pass proportion.

In this work, we strive to achieve code and functional coverage beyond 99% with suitable waivers. We define elaborate coverage collectors within the CPU monitor and SBUS UVC to capture the entire gamut of possible legal behaviors. Code coverage incorporates line, block, expression, FSM and toggle coverage.

4. FORMAL VERIFICATION

The class of formal methods encompasses several mathematical techniques which formally analyze the state space of a design [1]. A stark difference from simulation is that specific input stimulus is not supplied to the design under verification (DUV). In simulation, the test-bench continuously drives active values to the DUV, while monitoring the outputs. Therefore, a given execution can merely verify the design for that specific input combination. However, formal techniques ideally verify the design for all possible legal input values subject to assumptions, which constrain the inputs to legal behavior, and initialization, which is required to determine the legal reset states. This point is illustrated in Figure 4.1. Consider the circle to represent the legal state space of the DUV, simulation can merely provide spot coverage. State of the DUV is comprised of all state elements (flip-flops and latches), and inputs. Each simulation scenario when executed covers a single point on the design's state space. Formal analysis ideally promises full coverage of the entire problem space, but in reality due to capacity limitations, it is often only possible to validate critical subsets of the entire state space. This can be achieved by performing bounded model checking, abstraction, or other complexity reduction means. In this chapter, we briefly describe popular formal techniques, and their typical usage models. Then we proceed to explain our approach to formally verify the MESI cache design.

Broadly, formal methods can be categorized into three classes, namely:

Formal Property Verification (FPV): The use of formal tools to prove assertions identified from the specification. This technique, based on state space enumeration, is also known as model checking.

Formal Equivalence Verification (FEV): This technique is used to compare two models and determine their equivalence. A standard requirement in the digital design

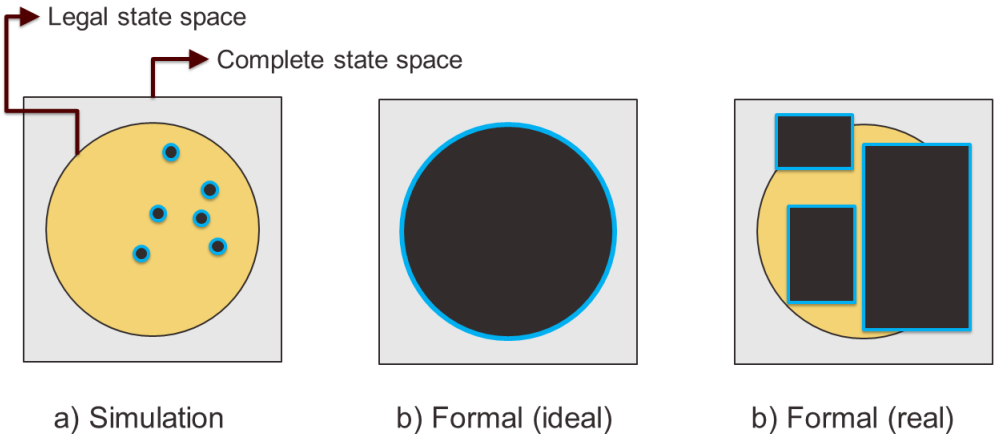

Figure 4.1: Motivation for formal verification

industry is to prove equivalence between the netlist and RTL or two different netlist versions.

Theorem Proving: This technique uses a set of specified, required properties (assertions) about a design to verify that these properties are satisfied by the implementation. A difference from model checking is that universal and existential qualification can be used to verify properties for any size of the design. Techniques used are also distinct; theorem provers use logic deduction, rewriting techniques and decision procedures to prove that the properties always hold true. Once these properties are verified, they become theorems about the implementation.

Model checking and formal equivalence verification are most relevant to the realm of hardware verification. Theorem proving techniques are sparingly used in digital design. In this work, we primarily focus on formal property verification of a MESI cache controller using model checking techniques. We begin with an initial specification and a toy RTL implementation of the multi-core cache, which is hereby referred to as initial design. We perform full proof FPV of the snooping cache design implemented in SystemVerilog

using efficient complexity reduction techniques. The initial design has several limitations like lack of parameterization, absence of a reset signal, and the presence of in-out ports. We overcome these limitations and other design flaws during the verification process to produce a robust design. Parameterization is critical to enable re-use, and modularity. Additionally, it facilitates structural abstraction, thereby permitting formal analysis of large, complex designs like a cache controller. In this chapter, we describe in detail the process and techniques used to make formal property verification of our RTL design feasible.

Formal equivalence checking is widely adopted during the synthesis of digital circuits. A downside of FEV is that it requires a reference model which we trust is correct and portrays the exact same behavior as the implementation (DUV). It is commonplace to witness FEV of an RTL model and a synthesized netlist. In this study, we adopt FEV to verify parameterization of our cache design. We demonstrate the use of equivalence checking as a useful tool during the front-end design phase. This effort is described in Section 4.2. Due to time limitations, we were unable to explore the uses and implications of theorem proving. Therefore, we include it as proposed future work in Section 6.1.

4.1 Formal Property Verification

Under this technique, design specification is described as a set of properties either in SystemVerilog or PSL (Property Specification Language). FPV, also popularly known as assertion-based verification (ABV), is essentially a method to prove that these properties hold for all legal input combinations and resulting reachable states for those inputs [31]. The generic flow of an FPV effort is indicated as Figure 4.2 [1]. The inputs to FPV are an RTL model, a set of properties to be proven, and a set of constraints. Assertions and cover statements form the set of properties which are required to be proven. An assertion is a statement about the design that is expected to always be true [1]. A cover point is a statement that describes an interesting condition or behavior about the design. It is

expected to be true occasionally, with a minimum of one occurrence. Assumptions along with clock and reset definitions constitute the set of constraints. Assumptions, similar to assertions, represent behavior that is universally true. In contrast to assertions, which describe DUV behavior, assumptions define the verification environment.

FPV uses a set of techniques to identify reachable states, and subsequently, verifies that the defined properties hold true in all reachable states. Traditional state enumeration technique uses an algorithm, like breadth first search, to calculate the reachable states. It continues enumeration until no additional states can be reached. Bounded model checking, on the other hand, uses different techniques. The most successful bounded model checking algorithm effectively converts the sequential design into a combinational circuit by unrolling it to bound k. Once unrolled, a satisfiability check is performed to determine if the properties are satisfied. Additionally, model checking uses satisfiability modulo theory (SMT) solvers, rewrite techniques and combinations of different algorithms. Post analysis, the model checking tool provides three plausible outputs for each property, namely proof, disproved/unreachable result, inconclusive or bounded proof. The list of proven properties includes proven assertions and unreachable cover statements. Every disproved assertion is supplemented with a counter-example waveform illustrating a failure scenario. Each reachable cover point provides a reachability trace describing the scenario in which the cover statement is achieved. An inconclusive or bounded proof indicates that the property was verified for a limited number of clock cycles from reset state.

Ideally, an FPV exercise aims to prove all assertions and strives to reach all valid cover statements. In such a scenario, engineers should review the proof environment to ensure that the design is not over constrained. In case of counter-examples and unreachable properties, manual debug effort is invested to identify the source of failure. Either the design is updated or constraints are refined to eliminate the failure. Nevertheless, inconclusive results are also useful as they signify partial success. If the bound on the result is sufficiently

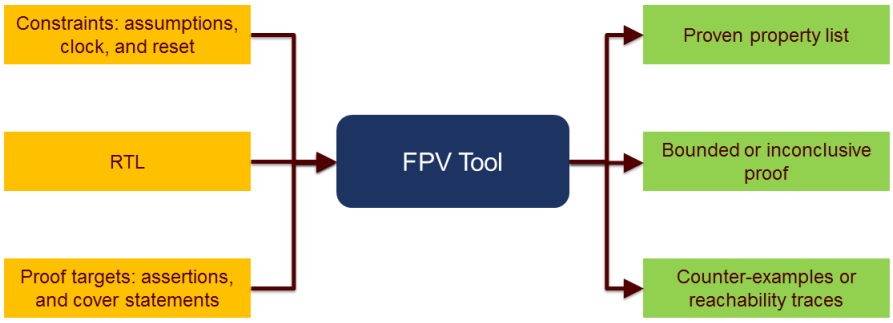

Figure 4.2: FPV tool execution. Reprinted from [1]

high, we can conditionally declare success as a counter-example could not be identified.

Property verification is not restricted to any given phase of the design flow. It serves as a generic tool providing numerous capabilities to interact with the implementation. Typical usage models of formal property verification are listed below [1].

1. **Early design exercise FPV:** Model checking can be used to gain insight into the initial functionality of the design. It can provide an instant test-bench to detect initial stage design flaws [32]. This exercise serves as a sanity check before the design is modified or released to the full fledged verification team.

2. **Bug hunting FPV:** Property verification is extremely powerful in detecting corner cases. Therefore, it could be used in collaboration with a simulation based approach. Specific high risk features are described as necessary properties and bugs are detected as counter example traces of the defined assertions.

3. **Full proof FPV:** Property verification could also serve as a complete replacement to simulation-based verification. In this usage model, the specification is fully defined as properties. Proof of the properties for all possible design states would imply that

the design conforms to the specification. This is the ideal usage model envisioned by the pioneers of formal verification. However, full proof FPV is infeasible for a majority of designs due to state space explosion and complexity.

4. **Specialized FPV:** Model checking is often applied to specific problems in the digital design cycle. A few examples include unreachable coverage elimination, connectivity [33], post-silicon debug [34], and protocol verification.

Our primary goal is to achieve full-proof verification of the cache module, with focus on the communication interface and coherence. Nonetheless, we explore opportunities to leverage other FPV usage models along the process of full-proof verification. Specifically, we also focus on FPV during bug fixes, unreachable coverage elimination, and FPV to understand legacy designs.

4.1.1 Verification Plan

A well-thought out plan is the first crucial step to a successful FPV endeavor. It is necessary to explicitly outline the objectives of the verification process. In this section, we describe our plan in terms of properties, verification levels, and completion criteria.

A brief summary of the formal verification plan is captured in Table 4.1. Our preferred tool for formal analysis is JasperGold from Cadence Design Systems. The goal is to verify coherence, consistency and the communication infrastructure. Cover statements are required to capture all documented transactions and basic behavior. Assumptions must limit the input space to legal combinations alone. Assertions, which are properties derived directly from the design specification, are primarily concerned with coherence and the input-output protocol.

The planned verification is staged into two-levels, namely uni-core and multi-core modules. Firstly, we define properties for the uni-core design with meaningful assumptions imposed on the inputs from other cores. Next, we extend verification to the multi-

Table 4.1: Formal verification plan

Goals	Verify cache coherence and memory consistency Verify the communication infrastructure (IO protocol)
Method	Formal property verification (model checking)
Properties	**Cover-points:** Typical behavior and transactions **Assumptions:** Constraints on the inputs Prevent illegal stimulus Represent entire range of legal behavior **Assertions:** Single-write multiple-read invariant Data value invariant Communication interface assertions Liveness guarantee
Staging plan	• Unicore module - Single cache line - Immediate bus access grant - Pseudo-LRU operation • Multi-core module
Complexity reduction	- Structural abstraction (parameterization) - Free variables - Memory abstraction
Exit criteria	Time limit Quality of proof Coverage Bugs discovered

core module by converting uni-core assumptions into assertions. This technique ensures that assumptions in the uni-core set-up do not over-constrain the design. The multi-core environment would still require some constraints on its inputs coming from level-2 cache. Additionally, verification of the uni-core module is staged in increasing order of complexity. Simplifications are assumed in the initial stages to enable easy debug. First, we prove the MESI invariants for a single cache line alone. Second, we do not consider the pseudo-LRU algorithm in our verification scope. Third, we assume that the arbiter would grant

bus access to our module immediately on the next cycle. These assumptions are excluded gradually as we make verification progress. Thus, the final stage of uni-core FPV does not include any presumptions.

The biggest threat to FPV is state space explosion. We employ effective complexity reduction techniques to tackle the state space problem. We leverage parameters of our design to reduce the size and make it tractable for JasperGold. Parameterized designs are ideal to apply structural abstraction [35]. Additionally, we also utilize free variables and memory abstraction within our formal analysis.

Simulation techniques usually decide process completion based on verification quality parameters like coverage, regression status, bug rate and quality, and pass percentage. Similarly, FPV completion can be analyzed as a combination of metrics. We set aside a fixed time limit for our FPV effort. This is common practice in the industry as teams need to meet aggressive time-to-market schedules. We analyze coverage, bugs discovered, and quality of proof to declare verification completeness.

4.1.2 Cover Statements

Cover points are useful to demonstrate that the proof environment can support typical behavior expected from the design. Statements would normally depend on the design-under-verification and its most critical functionality. However, common conditions listed below serve as good guidelines while defining effective cover points [1].

- Each waveform in the specification should map to a cover statement.
- Inputs and outputs must be capable of assuming all legal values in the environment.
- Every transaction type should be covered.
- Every state within the design is reachable.

- Easily identifiable error case scenarios should be included.

A complete list of cover points for our design, identified with the above directives, is provided in Appendix B.1. Abstract statements are converted into SystemVerilog cover points before they are fed into JasperGold.

Often, cover statements are incorrectly treated as an afterthought for FPV. This is a critical mistake as proven assertions do not have any significance if the FPV environment prevents basic operations. An over-constrained environment could disallow typical transactions on the DUV. Therefore, cover points are the primary focus during the initial stages of FPV. In the nascent phase of FPV, assumptions are refined through an iterative process, termed as wiggling the design. It is crucial to manually observe cover traces and confirm functionality during the wiggling process. Subsequently, cover traces should be analyzed every-time there is a change to the set of constraints. We rigorously follow the aforementioned recommendation to ensure sanity of our FPV environment at every stage.

On the other extreme, engineers with a strong simulation background could easily develop an affinity to cover statements, as they are strikingly similar to test scenarios. This situation is equally undesirable as reachable statements can only guarantee a possibility of good behavior. They do not verify good behavior under all circumstances. Therefore, we should prevent cover statements from turning into a set of simulation test cases. The ultimate focus of FPV is to prove assertions. This further reinforces the belief that formal analysis requires a fundamentally different perspective when compared to simulation.

4.1.3 Complexity Staging

We subdivide the verification process into two levels, namely the uni-core module and the multi-core module. As described earlier, the many-core module consists of multiple instantiations of the uni-core block. First, we define a complete set of properties for the uni-core module. Several assumptions are imposed to limit the input stimulus to legal

combinations. Once cover statements and assertions are proven at the single core level, it is crucial to validate the constraints utilized for FPV. If the set of assumptions over-constrain the design space, our proof is incomplete. Constraints can be justified in several ways. We could provide arguments from the input specification. Alternatively, we could assert the constraints in a simulation regression suite, typically at a higher level of hierarchy (IP/SoC level). In our case, we simply convert the uni-core assumptions into internal assertions in the multi-core environment. As a result, the assumptions are automatically validated at the next level of verification. Constraints in the multi-core environment, which are concerned with signals coming from the level-2 cache and arbiter, are justified using arguments from the design specification. All assumptions utilized in our FPV exercise are described in Appendix B.2. We verify that all assertions are valid in the multi-core model as well. This is necessary to reveal any concurrent behavior that was not present in the uni-core model. Additionally, the multi-core model may contain deadlocks or livelocks which are not present in the uni-core model.

Techniques to handle complexity in large designs can be classified into either under-constraints or over-constraints. Under-constraining the design makes it more generic by abstracting away intricate details. With a more generic design, we run the risk of false negatives which demand greater debug effort, but false positives are impossible. Over-constraining, on the other hand, makes the design more specific. Therefore, it can lead to false positives. In other words, there is the possibility of a faulty design being declared correct and functional. As a result, over-constraining the model is extremely risky and needs to be justified.

Nonetheless, it is straight forward to debug and understand a simplified design, especially during the early stages of verification. A simple model with a limited set of behavior can be obtained with over-constraints. Once we comprehend and debug a variety of transactions, we can gradually eliminate certain constraints to extend support for

more complex behavior. Strict adherence to LRU operation is not critical to coherence or consistency. Therefore, we exclude verification of pseudo-LRU functionality in all stages to simplify our FPV effort. We execute uni-core module FPV in the below listed stages, which increase in level of complexity.

1. *Stage-1*

   In the simplest stage of FPV, we prove MESI transition assertions for a single cache line. We assume that system bus access is granted to the module immediately. Writes are restricted to data blocks alone. Additionally, a invalidate request is also serviced immediately on the next clock cycle. We assume that data will be provided on the shared bus within a fixed latency of 2 clock cycles. Assumptions specific to stage-1 are listed in Appendix B.2.1.1.

2. *Stage-2*

   In stage-2, we allow write operations to instruction cache, and verify that they are not performed by the implementation. We prove the MESI assertions for all cache lines. We increase the bus latency to 3 cycles. Assumptions specific to stage-2 are listed in Appendix B.2.1.2.

3. *Stage-3*

   In stage-3, we describe arbiter and level-2 behavior more accurately. We assume that a level-1 cache is granted primary system bus access within 45 cycles. We arrived at an estimate of 45 cycles, based on the realistic observation that each level-1 request can take a maximum of 15 clock cycles. This is the theoretical worst case estimate for a 4-core system, in which all other level-1 caches perform eviction and write back before access is granted to our L1-cache of interest. Additionally, we place a realistic assumption that snoop access is granted only for blocks present in modified or exclusive states. Snoop grant may or may not be granted for a block in shared

state. Based on observations in the multi-core FPV, we placed modified the bus latency to a maximum of 9 clock cycles. Assumptions specific to stage-3 are listed in Appendix B.2.1.3.

Clearly, the final stage of the uni-core FPV does not permit any over-constraining as stage-3 constraints are realistic in the final implementation.

4.1.4 Assertions

The crux of formal property verification is the set of assertions. Assertions are derived directly from the design specification. Goals of the FPV exercise dictate the number and type of assertions. A design sanity FPV would be complete with limited assertions ensuring basic behaviors alone. Similarly, it would suffice to describe risky features as properties for bug hunting FPV. However, a full-proof verification is expected to completely replace simulation. Therefore, the complete specification is required to be captured as SystemVerilog assertions.

Since the simulation effort preceded formal verification, we used assertions identified for simulation as the starting point for FPV. Assertions from the UVM environment were limited and mostly unsuitable for FPV. Hence, a significant effort was needed to develop properties appropriate for FPV. It is occasionally acceptable to define assertions with infinite length sequences in simulation, but they are unsuitable for formal property verification due to state space explosion. Similarly, MESI transition, memory coherence, and some interface assertions are not required in the UVM environment due to the presence of a reference model and elaborate monitors.

The set of properties used for our FPV is presented as Appendix B.3. We classify proof targets based on the specific feature that they represent. We have the following categories of assertions:

1. CPU-lv1 interface

2. System bus interface

3. Bounded liveness properties

4. MESI state transitions

5. Coherence and memory consistency

6. Bug fix assertions

Appendix B provides complete detail of the covers, assumptions, and assertions. FPV properties are stored in a separate module known as the verification component (VCOMP). It is basically a SystemVerilog module with only input ports. All the cover statements, assumptions and assertions are defined within this module in terms of its inputs. Finally, the VCOMP is bound to the relevant design module with the 'bind' directive in SystemVerilog. Verification components separate FPV properties from the design. We can easily void the bind directive in synthesis or simulation. Assertions embedded in the design, when large in number, can delay simulation runs significantly. This is avoided with the practice of encapsulating properties in a verification component.

4.1.5 Complexity Reduction Techniques

Formal techniques are inherently solving an NP-complete problem. Consider a design with n state elements, which are either flip-flops or latches, and m input signals. State of the design is comprised of both state elements and inputs. Theoretically, it could have $2^{(m+n)}$ possible configurations in its state space. Therefore, even for reasonable values of m and n, the formal task is challenging. Exponential growth in state space with increase in design size is the main source of complexity in formal analysis. However, in practice, only a portion of the entire design space can be reached from the reset state. Additionally, the FPV tool only needs to analyze state in terms of the logic cone, also known as Cone

of Influence (COI), of the property. These factors along with numerous advances in recent years help enable formal analysis of large designs.

Despite improvements in formal technology, state space explosion still presents a major hindrance to full scale adoption of formal methods [1]. Complexity issues are easily identifiable. They typically show up as tool crashes, time-outs, memory blowups, and bounded results. Clever techniques like abstraction, black-boxing, and cut-points are required to navigate complexity. In this section, we discuss techniques used to overcome complexity issues encountered during FPV of our cache design.

4.1.5.1 Formal friendly properties

It is easy to describe the same property in numerous ways using SVA. Unlike the Python programming language, SystemVerilog assertions allow various options to describe the exact same behavior. In order to facilitate formal analysis, we ensure that properties are simple and small. Here, we describe specific examples of formal-averse and formal-friendly properties for equivalent behavior.

Consider the properties shown below:

```
1 A1: assert property ((a || b) |-> c);
2 A2: assert property (d |-> (e && f));
```

We can easily re-write the above properties as multiple smaller assertions shown below. This is most helpful when a, b, e and f are not boolean signals, but complex sequences.

```
1 F1_1: assert property (a |-> c);
2 F1_2: assert property (b |-> c);
3 F2_1: assert property (d |-> e);
4 F2_1: assert property (d |-> f);
```

On several instances in the uni-core module, we are required to assert that signal x remains logic high until a rising edge of signal y is observed. We illustrate the formal-averse property below:

```
1 A1: assert property ($rose(x) |=> x until $rose(y) );
```

Although the above property is natural and easy to understand, 'until' keyword in

SystemVerilog leads to problems in FV. Most simulation engineers are accustomed to describe properties in the above coding style. We can represent the same behavior with a formal-friendly, boolean assertion as shown below:

```
1 F1: assert property (x && !y |=> x);
```

Similarly, in order to ensure that data is unchanged as long as data_valid remains high, we can easily avoid the until directive by replacing A1 with F1 as shown below:

```
1 A1: assert property ($rose(data_valid) |=> $stable(data) until $fell(data_valid));
```

```
1 F1: assert property (data_valid |=> (!data_valid || $stable(data)));
```

Liveness properties are the hardest to prove in FPV. Typically, they result in infinite length sequences which triggers the state explosion problem. Therefore, it is immensely helpful to re-write liveness using finite length properties as real world systems are expected to respond in finite time. In order to guarantee that every CPU read operation eventually obtains a response, we could inefficiently define an infinite length assertion as shown below.

```
1 A1: assert property ($rose(cpu_rd) |-> s_eventually(data_valid));
```

Analysis is easier for the FPV tool if we guesstimate a worst case latency for the CPU read operation as described below:

```
1 F1: assert property ($rose(cpu_rd) |-> ##[1:50] $rose(data_valid));
```

In summary, we followed the below principles to enable easier analysis:

1. Divide complex behaviors into small, simple properties
2. Perform boolean simplifications whenever possible
3. Avoid directives like until, s_eventually, and throughout
4. Make liveness properties finite

Inefficient coding does not merely apply to assertions alone. Assumptions, which are coded in a formal-averse manner, can cause greater damage as they appear in the fan-in of several assertions. A poorly coded assertion affects only itself, but a badly written assumption can affect several properties.

4.1.5.2 Auxiliary code

Assertion languages have limited scope and expressiveness. It is impossible to define overlapping behavior in SVA. For instance, consider the requirement that every pulse on a 'request' signal should be acknowledged by a corresponding pulse on signal 'ack' within 4-8 cycles. We cannot define assertions to fully capture this condition. Therefore, we utilize auxiliary SV code which keeps track of pending requests. Supplementary SV code is embedded in the verification component module in order to keep it isolated from the design.

Additionally, certain behaviors are extremely complex to be described in SVA alone. FPV can benefit immensely with the additional of SV code for such scenarios. Auxiliary code simplifies the properties, making analysis easier for the FPV tool. Therefore, we often use additional SV code within our verification component to facilitate FPV. Typically, auxiliary code is in the form of finite state machines and counters. One particular example is the behavior of the round-robin arbiter in our system. Assumptions which characterize the behavior of our round-robin arbiter, are possible only with auxiliary code which remembers the number of grants provided to other L1 caches. Reference models for data coherence, described in the next section, are a special case of auxiliary code.

4.1.5.3 Reference models

Reference models convert a portion of the FPV exercise into formal equivalence verification. Complex assertions, defined on internals signals within the DUV, are transformed into a direct comparison between the RTL and the reference model. As a result, they

increase the capacity of the FPV tool.

Within the purview of our FPV exercise, we define a reference model to assist the coherence assertions described in Appendix B.3.1.5. This model simply maintains a copy of valid data for the entire address space. Within the assertions, it suffices to compare our copy of valid data with the model outputs. Despite its benefits, we should exercise restraint and avoid excessive use of auxiliary code. It is inefficient to recreate large, intricate portions of RTL within the verification component. With excessive SV code, we run the risk of false positives, due to bugs present in both the reference model and the RTL.

4.1.5.4 Parameterization

Structural abstraction is the most significant and straight forward method to reduce complexity. If the design is parameterized, we can easily simplify a large data-path or complex structure. Consider the case of a wide data bus with the same logic for every data bit. We can minimize the width to a single bit and still maintain confidence in FPV.

Within the purview of our design, we exercise SystemVerilog parameters to make FPV feasible on our implementation. We reduce the address width to 7 bits, L1 cache size to 64 bits, number of cache sets to 4, and address tag to 3 bits. This decision was the biggest enabling factor for our FPV effort. Almost all the properties were inconclusive when FPV was performed on the full-size design. Despite the massive reduction in design size, we can theoretically argue that all behaviors possible in the original size design are retained in the reduced model. For instance, if we minimized the design to consist of single cache set, we could clearly point out complex behavior that is improbable in the reduced model. Parameterization can be made complete by using induction to prove from the base case, that the arbitrary general parameterized model is correct.

4.1.5.5 Free variables

Free variables, similar to primary inputs, can be assigned arbitrary values by the FPV engine subject to imposed constraints. A free variable is typically added to facilitate formal verification when the model contains significant amount of symmetry [1]. In such a scenario, we can exploit free variables to generalize formal analysis. Different cases exercising similar logic is combined and verified by a single assertion rather than separate properties.

In our FPV effort, we introduce a free variable to represent cache line number, by declaring a new variable within the verification component. Without the concept of free variables, MESI protocol assertions are realized with the help of a generate statement as shown below:

```
1  parameter NUMBER_OF_CACHE_LINES = 16;
2  generate for (genvar i=0; i<NUMBER_OF_CACHE_LINES; i++) begin
3          A1: assert property MESI_assertion(i);
4          ....
5          end
6  endgenerate
```

The above code snippet would create 16 instances of MESI assertions, one for each cache line. Similar logic is verified in slightly different versions of the same assertion, resulting in a lot of inefficiency. With the addition of a new free variable, we can generalize analysis for all the cache lines. Therefore, we rewrite assertion A1 as shown below to exploit symmetry within the design.

```
1  parameter NUMBER_OF_CACHE_LINES = 16;
2  int free_i; //free variable
3  U1: assume property (free_i >=0 && free_i < NUMBER_OF_CACHE_LINES);
4  U2: assume property (##1 $stable(free_i))
5  A1: assert property MESI_assertion(free_i);
6  ....
```

Here, variable 'free_i' can take any of the 16 legal values, thereby forcing a broader analysis of the assertion. The FPV tool can exploit symmetry and similarity in logic to prove the assertion for all cache lines at once. Restrictions on the free variable are extremely significant and need to be justified. Here, we simply impose constraints on the

new variable to take on legal values and remain constant throughout a trace execution. Nevertheless, introduction of free variables is a double-edged sword. If we combine completely dissimilar logic with the help of a new variable, tool efficiency would decrease significantly.

4.1.5.6 Memory abstraction

Large memories, queues and counters are the most problematic structures for formal verification. They are often replaced by abstract models to facilitate effective analysis. Black-boxing, cut-points, and free variables can be considered simple forms of abstraction. Essentially, we retain only a significant portion of the model, while abstracting away intricate details.

Our cache implementation has a large memory, but this is already handled with parameterization. However, within the verification component, we are required to accurately recreate behavior of the level-2 cache and main memory. Even in the minimal system, level-2 cache is required to hold values of 32 blocks each 4-bit wide. We cannot obtain FPV proof results without an accurate assumptions on outputs of the level-2 cache. This is made possible through simple memory abstraction technique. Auxiliary code within the verification component adds to state space problem as well. Thus, memory abstraction of level-2 cache is crucial to our FPV effort.

We replace a full-capacity level-2 model with a small set of interesting locations. Using free variables, we restrict access to a maximum of ten different locations in one execution-trace. This significantly reduces complexity as we track only 10 blocks within the verification module. This is a reasonable assumption, as no interesting behavior is excluded. In general, abstraction is not a fool proof method, it is capable of over-constraining the design, which in turn can lead to false positives.

4.2 Formal Equivalence Verification

Formal equivalence verification (FEV) is the mathematical process of proving two distinct models to be logically equivalent. These two models are typically referred to as the specification (SPEC) and the implementation (IMP). Specification, generally more abstract as compared to IMP, is either an RTL description, a high-level model, or a synthesized netlist. Implementation, which is more concrete, could be an updated RTL specification or an optimized netlist. Several notions of equivalence exist depending on the points of comparison between the two models. Aspects of the two modules which form the basis of comparison are known as key-points. Common key-points include inputs, outputs, state elements, and cut-points [1].

On the basis of the definition of equivalence, FEV can be classified into the following categories [1]:

1. *Combinational Equivalence:*

   Combinational equivalence, also known as state-matching equivalence, is the most mature FEV technique [36]. Key-points for this method include inputs, outputs and all state elements. The two models are expected to have equivalent internal states given the same input stimulus. This is easiest form of equivalence in terms of analysis, as the tool can treat state elements as cut points. It is only required to compare combinational logic between flip-flops and latches. A huge limitation of state-matching is that even minor changes in structure can disrupt equivalence. Despite this limitation, combinational equivalence is the norm in synthesis of digital designs. The synthesized netlist is expected to be state equivalent to the RTL implementation.

2. *Sequential Equivalence:*

   Two models are known to be cycle-accurate or sequentially equivalent if their out-

puts match on every clock cycle given the same set of inputs [37]. Key-points for comparison are comprised of only primary outputs. Despite changes to state representation, pipeline depth, and internal timing protocols, sequential equivalence can verify equivalent designs.

3. *Transactional Equivalence:*

   Transactional equivalence is a relatively nascent technology used to compare between a highly abstract model and an RTL implementation. The high level model is not cycle-accurate but rather transaction accurate [38]. In other words, comparison is performed at the boundaries of defined transactions. This definition of equivalence is more generic as compared to earlier notions. However, it is also the most challenging form of FEV.

At its core, FEV is similar in several aspects to FPV. In the absence of a dedicated FEV tool, we can perform equivalence verification by defining properties that assert equivalent key-points. The typical FEV execution flow is depicted in Figure 4.3. Inputs to the FEV tool include input constraints, the two models for comparison and key-point mappings. Outputs from the FEV tool are one of three possibilities, namely proven equivalence, inconclusive, or an inequality. If inequality is established, a counter-example is provided for the scenario that captures the difference.

Common usage models for FEV are described below [1, 39]:

- RTL vs netlist equivalence

  During logic synthesis, the synthesized netlist is often compared against the original RTL for functionality. Combinational equivalence with is generally used for this exercise.

- Verification of parameterization

  Adding configurable parameters to a legacy design is a risky affair. Designers of-

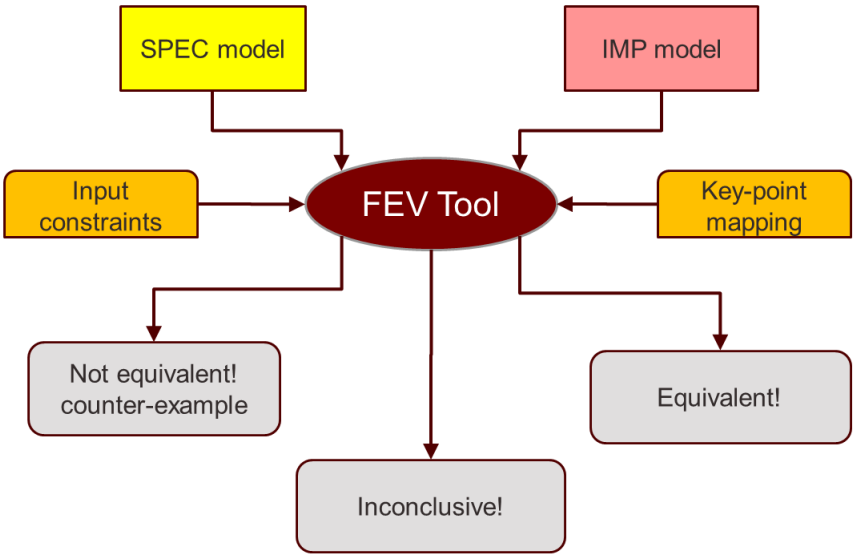

Figure 4.3: Typical FEV execution

ten fear that basic functionality could be disrupted in the process. Combinational FEV is an easy check to ensure that functionality in the default configuration is preserved. The non-parameterized model is compared against the parameterized model in default configuration. However, FEV does not comment about functionality in a new configuration. In other words, it does not guarantee that parameterization is complete and successful. Induction proofs can be used to completely verify parameterization.

- Validating timing fixes

  Timing changes are often performed late in the design cycle. Techniques to ease timing include logic distribution, and critical path reduction. Simple sequential FEV allays fears of a broken design.

- Chicken bit validation

  Late in the design process, de-feature (chicken) bits are added to disable design

changes with low confidence. FEV can provide peace of mind by proving equivalence between the stable design and updated design with chicken bit enabled.

- Clock gating verification

  Clock gating is an important power reduction technique, which is capable of disrupting functionality late in the design process. Proving equivalence between original and gated implementations prevents the addition of undesirable failures.

With reference to our study, we could not comprehensively investigate the application of FEV techniques to our design due to limited resources and time. However, as a simple case study, we proved equivalence between the initial unparameterized design and the updated, configurable implementation. The legacy design did not facilitate a flexible number of cores. We added this feature to the module and ensured equivalence for the default configuration of 4 cores. This was performed using the JasperGold Sequential Equivalence Checking (SEC) application. Although this exercise was trivial enough to be realized in a few minutes, it demonstrates the effectiveness of FEV during design updates as described earlier in this section. It reinforces our belief that formal techniques serve as a generic tool-kit with numerous opportunities to interact with the RTL implementation.

5. RESULTS

In this chapter, we present results obtained using the two verification techniques. We describe the metrics used to analyze progress in simulation and formal methods. Additionally, critical design changes made to fix deviations from the specification are reported under each technique. A vital coherence issue, due to contention between CPU and snoop-side requests, was identified in formal analysis. This led to a significant revamp of the micro-architecture, which is detailed in Section 5.2.2. The tools, and infrastructure used in each approach are summarized in Table 5.1.

Table 5.1: Infrastructure for verification

	UVM Environment	**Formal Verification**
Language	SystemVerilog and UVM	SystemVerilog Assertions
Tool(s)	Cadence Irun SimVision VManager Incisive Metrics Center	Cadence JasperGold FPV App Cadence JasperGold SEC App
Machine	Dell PowerEdge R815 4 AMD Opteron 6174 Processors (48 2.2 GHz cores) 256 GB main memory CentOS 5.7 x86_64	

5.1 UVM Environment

The system parameters used within the UVM environment is the original design intent, which is 32-bit address, 32-bit instruction and data blocks, and a 4-core configuration. Level-1 cache has the following settings: 16-bit address tag, 14-bit set index, 2-bit offset, 2 Kilo-Bytes each for data and instructions, and 4-way set associative. Level-2 cache is 8-way set associative model with a unified capacity of 8 Mega-Bytes. However, a reduced

size configuration, equivalent to that used in the formal technique, is also analyzed in the UVM environment. This is possible due to the flexible nature of our UVM test-bench. It ensures that functionality is not lost in the reduced configuration. Results obtained from the minimal parameters are clearly specified with the label 'RC'.

Table 5.2: Time-line of verification progress

Date	**Infrastructure developed (%)**	**Test-cases developed**	**Test-cases passing**	**Pass rate (%)**
28-Oct-16	0	0	0	N/A
04-Nov-16	20	0	0	N/A
11-Nov-16	40	6	3	50.00
12-Nov-16	70	15	8	53.33
13-Nov-16	75	16	16	100.00
25-Nov-16	80	16	16	100.00
01-Dec-16	95	16	16	100.00
02-Dec-16	100	21	18	85.71
04-Dec-16	100	22	21	95.45
08-Jan-17	100	22	21	95.45
10-Jan-17	100	22	22	100.00
14-Apr-16	100	22	21	95.45
12-May-17	100	22	21	95.45
19-May-17	100	22	22	100.00

We successfully developed the necessary infrastructure and executed the verification plan outlined in Chapter 3. The execution timeline is presented as Table 5.2. It describes the number of test-cases coded and passing with a single fixed seed. Additionally, an estimate of infrastructure development is provided. At the close of our project, we were able to execute all test scenarios without any failures.

We developed a regression suite consisting of 22 test-cases, each verified over hundred random seeds. Progress in the regression suite is depicted in Tables 5.3 and 5.4 for the original and minimized (RC) configurations respectively. A graphical illustration of the

Table 5.3: Status of regression suite for original design parameters

Date	Test-cases run	Test-cases passing	Pass rate (%)
11-Nov-16	6	3	50.00
12-Nov-16	180	167	92.78
13-Nov-16	320	320	100.00
01-Dec-16	1600	1600	100.00
02-Dec-16	2100	1987	94.62
04-Dec-16	2200	2119	96.32
08-Jan-17	110	97	88.18
10-Jan-17	110	104	94.55
12-May-17	440	407	92.50
19-May-17	440	440	100.00
26-May-17	2200	2200	100.00
02-Jun-17	2200	2200	100.00
09-Jun-17	2200	2200	100.00

Table 5.4: Status of regression suite for reduced design parameters (RC)

Date	Test-cases run	Test-cases passing	Pass rate (%)
14-Apr-17	440	414	94.09
12-May-17	440	418	95.00
19-May-17	440	440	100.00
26-May-17	2200	2200	100.00
02-Jun-17	2200	2200	100.00

pass rate is provided as Figure 5.1. We notice the presence of a blocking bug in early December 2016, which led to several regression failures. The source of these failures was identified to be contention between the processor and snoop-side requests. Simultaneous requests to the same address block from the CPU and system bus, led to several failures. In order to diagnose and device a solution to this concurrency issue, we reduced the number of random seeds to 5, resulting in a total of 110 test cases in the suite. Although, this issue was identified in simulation, we failed to define a suitable solution in spite of numerous attempts. We were able to reduce the failure rate occasionally, but failed to eliminate

the error completely. This was because errors cropped up only in random tests, which had a substantially large simulation time. It was challenging to isolate the failure causing transactions from irrelevant transactions. Finally, a definitive solution was identified in formal analysis, as described in Section 5.2.2. Once this fix was implemented, we switched to hundred random seeds and obtained complete success in our regression suite.

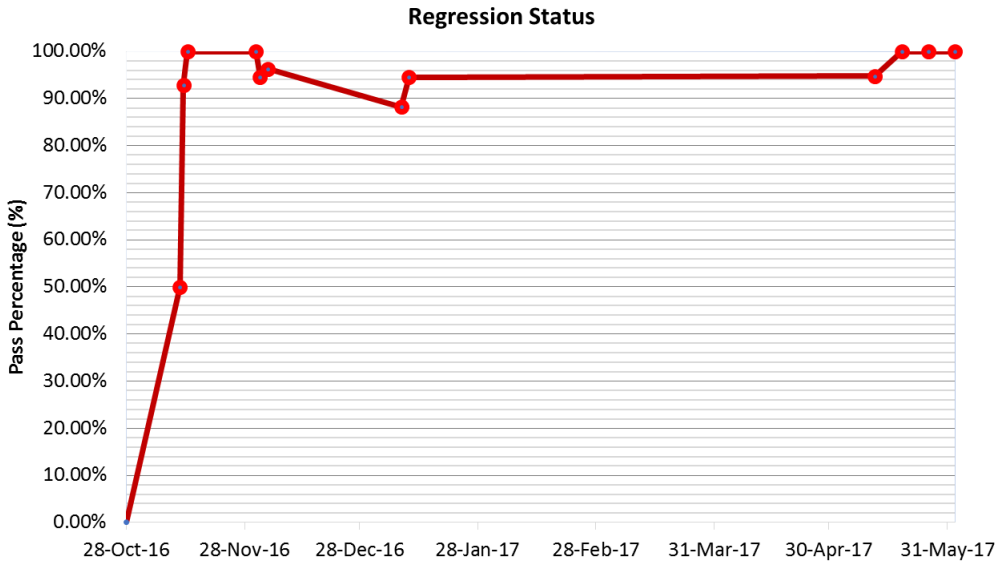

Figure 5.1: Pass rate based on regression suite

Critical bugs identified and fixed in simulation are described in Section 5.1.1. Thereafter, coverage metrics are provided in Section 5.1.2.

5.1.1 Bugs Discovered

A total of 13 bugs were discovered, and 12 were fixed using the UVM simulation framework. An important point to note is that preliminary verification was already performed on the legacy design, similar to how designers sanity check RTL prior to release to the verification team. As a result, most trivial bugs were eliminated before creation of

the UVM framework.

5.1.1.1 Signal cp_in_cache de-asserted during bus_rd_snoop

Consider a scenario in which data block A is present in L1-cache0, which services CPU0. In response to an incoming snoop side bus_rd_snoop request from L1-cache1 for block A, L1-cache0 should respond by asserting signals cp_in_cache and shared_local. Signal shared_local was asserted but cp_in_cache remained low. This is corrected by a minor fix to the RTL of the main functional cache block.

5.1.1.2 Signal cp_in_cache de-asserted during bus_rdx_snoop

Consider the behavior in the previous bug, but for an incoming bus_rdx_snoop request This flaw is fixed by a minor correction to the RTL.

5.1.1.3 Signal shared_local not generated for modified snoop block hit

Imagine a situation in which data block A exists in modified state in L1-cache0. A read operation by CPU1, initiates a bus_rd request from L1-cache1. L1-cache0 observes the bus read request and performs write-back of block A. Once level-2 write is successful, L1-cache0 should supply block A to L1-cache1 with the shared_local signal enabled to indicate that the block is shared. However, the implementation incorrectly drives the shared_local signal low when data is provided to L1-cache1 on the system bus. A simple fix resolved the issue.

5.1.1.4 Incorrect LRU replacement

When the LRU state variable is 3'b1x1, way-3 should be replaced from the cache set, according to the specification. Our legacy implementation incorrectly evicts way-2 of the cache set.

5.1.1.5 Incorrect instruction address bound

Our implemented incorrectly treated address 32'h3FFFF_FFFF as a data access. The specification dictates that this is a instruction address. This was caused by the use of a wrong relation operator.

5.1.1.6 MESI state update during invalidate_snoop

Consider a situation in which two level-1 caches (0 and 1) have a data block A in shared state. When CPU1 performs a write operation to block A, L1-cache1 would send out an invalidate request on the system bus. The incoming invalidation message is processed by L1-cache0, but the MESI state of block A is not changed to invalid. Internally, the MESI state was updated by the wrong variable. This is corrected by fixing the internal variable.

5.1.1.7 LRU eviction does not invalidate cache line

In the case that a cache set is full, an incoming CPU operation to the same set, which results in a miss, would force eviction of a cache line. If the evicted cache line is in modified state, dirty data is written to level-2 cache, but the cache line is not invalidated. As a result, the CPU operation is not successful. Behavior was similar for eviction of a shared block. Once again, the cause of this bug was a faulty internal variable.

5.1.1.8 Bus request dropped after LRU eviction

The specification states that LRU replacement and the bus request should be performed as a single operation on the system bus. Our legacy design drops the request for primary bus access soon after the eviction is completed. A new request for the common bus is initiated later to receive the CPU requested data block. This is a clear deviation from the specification, which is rectified with minor corrections to the RTL.

5.1.1.9 Response to snoop-side invalidate request

Consider a block A to be present in shared state in 3 level-1 caches (0,1 and 2). A write request from CPU2 would generate an invalidate on the system bus from L1-cache2. Our legacy design is tuned in such a way that both L1-cache0 and L1-cache1 respond by asserting their own invalidation_done signals for exactly one clock cycle. In most scenarios, L1-cache0 and L1-cache1 respond at the same clock (immediately), therefore error is masked. If either of the caches took longer to respond, then all_invalidation_done signal is never generated. A late response is possible during corner cases such as a parallel conflicting CPU request. Fortunately, we identified this error in one of the pseudo-random test-cases. We fixed the issue by forcing the uni-core module to assert the invalidation_done signal until all_invalidation_done is observed as logic high.

5.1.1.10 Discrepancy with reference model due to silent eviction

This discrepancy between the cache reference model and the implementation is due to the non-atomic nature of state transitions. Cache reference model assumes that each CPU operation is atomic, and concurrent CPU operations are executed in a fixed order as received on the CPU monitors. Arbiter in the design should essentially ensures that any conflicting operations are serialized. We illustrate the issue with the help of an example as shown in Figure 5.2.

Consider 5 data blocks (A1, A2, A3, A4, and A5) belonging to the same set A. L1-cache0 has blocks A1, A2, A3, and A4 in shared, exclusive, modified and modified states respectively. L1-cache1 receives a CPU read for block A2. It sends out a request for primary access (proc) to the system bus. Once granted primary access to the common bus by the arbiter, L1-cache1 sends out a bus read request for block A2. Simultaneously, CPU0 transmits a read request for block A5, which triggers silent eviction of block A2 due to the LRU replacement policy.

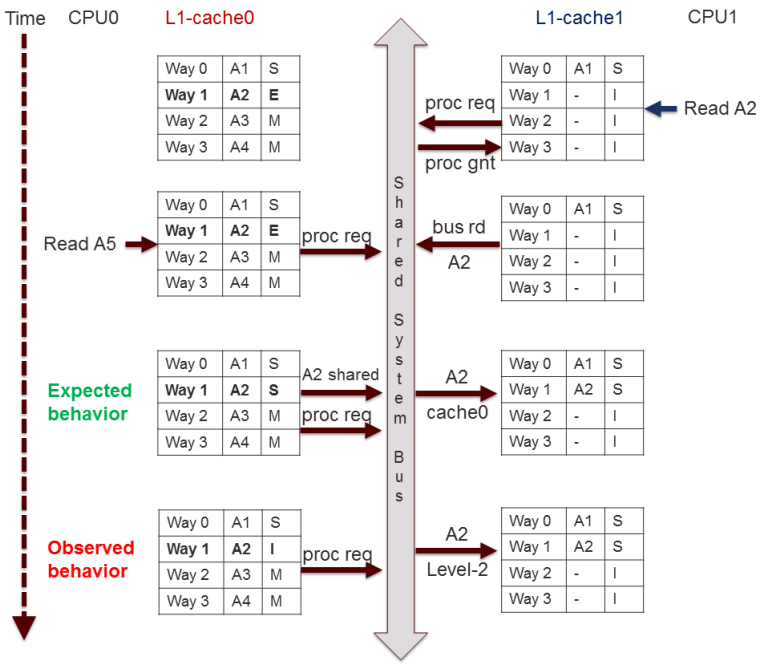

Figure 5.2: Silent eviction issue

Our reference model dictates that the bus read to A2 is serviced by L1-cache0. It assumes that as CPU read to A5 is not completed, eviction of A2 should not be done yet. However, our design does not wait for bus grant before silently evicting a block in shared or exclusive state. Therefore, bus read os A2 is serviced by level-2 cache. This discrepancy is fixed by simply waiting for primary grant (proc grant) of the system bus before performing silent eviction. This error cropped up only in tests with significantly random stimulus. Debug effort required was high due to the waveforms with considerably large simulation time.

5.1.1.11 Incorrect update of LRU state variable

The LRU state variable should be updated whenever a CPU operation is performed on the cache set. Consider two sets A and B; block A1 resides in way-1 of set A and block B2 is present in way-2 of set B. The CPU performs a read to B2, quickly followed by a

read to block A1. The LRU logic in our implementation updates its state whenever there is a change in either block_accessed or index_proc, both of which are internally generated signals. Signal block_accessed is the output of a register while index_proc is mere combinational logic. As a result, there is a single cycle delay between block_accessed and index_proc. This delay causes the LRU state of set A to be updated as if way-2 was being accessed before way-1 is looked-up. The expected behavior is that LRU state is updated only considering access to way-1. This leads to incorrect LRU state.

This behavior manifested as the unnecessary eviction of a cache block several thousand cycles after the actual bug. It was challenging to identify the source of faulty behavior. Several design changes across the complete hierarchy were necessary to arrive at a working fix. The IO ports of numerous internal modules were changed. Therefore, we consider this bug a serious flaw, leading to a major design change. Finally, our fix ensured that LRU state was updated only when a new internal signal lru_update was asserted. 'lru_update' went high for exactly one cycle when the block being accessed is hit within the cache.

5.1.1.12 LRU state variable for shared to modified transition

This error is the result of the fix suggested in Section 5.1.1.11. Several test-cases after the fix was made, we realized that the resolution was incomplete for a very specific scenario. This corner case bug was extremely challenging to narrow down with each failed waveform running for at least a few hundred thousand clock cycles. Although the LRU state is updated incorrectly at a relatively early stage, it takes a lot of time before the bug manifests itself as a failure.

We realized that for a CPU write operation, which triggers a shared to modified transition, internal signal lru_update is incorrectly generated multiple times. We fixed this error by aligning lru_update to the actual CPU response.

5.1.1.13 Contention between CPU and snoop-side requests

The biggest challenge we faced in simulation was the contention issue between CPU and snoop-side requests. Several failures within the regression suite were associated to this issue. We noticed that whenever conflicting requests to the same block were made on the CPU and snoop-side concurrently, several failures would be observed. Essentially the uni-core module attempts to service both the CPU side request and the snoop-side operation in parallel, which leads to unpredictable behavior. A simple example is a CPU write to block A, which is present as modified in the cache, and a parallel bus_rdx message on the system bus. As the cache attempts to service both requests, coherence and consistency are effortlessly lost.

Although we narrowed down the source of failure, we failed to identify a complete solution using the simulation framework. We attempted several fixes, but we could merely increase the pass rate in regressions. We never managed to eliminate the issue completely. Different fixes resulted in special corner-case errors, which took several days to analyze. Each failing trace was at least five hundred thousand nanoseconds of simulation time with numerous concurrent transactions. We gave up due to the sheer time and effort required for analysis. Finally, we were able to resolve the issue with much ease in formal property verification. We discuss this issue and its fix at greater length in Section 5.2.2.3.

5.1.2 Coverage

Coverage metrics obtained using the final regression run, with all cases passing, are presented in Tables 5.5, 5.6, and 5.7.

We went through a few iterations to refine functional coverage by excluding unreachable cover points, after which we attained complete functional coverage. However, code coverage values presented are unadulterated, without any waivers or exclusions. Refined metrics obtained after manual analysis is presented in Section 5.4.3. Verification engineers

Table 5.5: Code coverage metrics without any exclusions

Instance	Block	Expression	Toggle
Multi-core L1 cache	96.47%	93.69%	93.76%
Uni-core L1-cache(i)	96.47%	93.69%	89.49%
L1-cache(i) data cache	96.96%	99.11%	93.77%
L1-cache(i) inst cache	95.99%	69.44%	84.29%
L2 cache	83.08%	100%	65.03%
Arbiter	95.24%	-	100%
Memory	71.43%	-	53.25%

Table 5.6: Assertion coverage metrics

Type	Coverage
CPU0-lv1 interface	100%
CPU1-lv1 interface	100%
CPU2-lv1 interface	100%
CPU3-lv1 interface	100%
System bus interface	100%
FPV assumptions	100%

Table 5.7: Functional coverage after analysis

Type	Coverage
CPU0 monitor	100%
CPU1 monitor	100%
CPU2 monitor	100%
CPU3 monitor	100%
System bus monitor	100%
MESI coverage	100%
LRU coverage	100%

in the industry often spend weeks in analyzing uncovered portions of coverage. We plan to execute this time consuming analysis of code coverage in collusion with a formal tool called JasperGold Unreachable Coverage (JG UNR), which promises to simplify the ef-

fort. Our goal is to evaluate the applicability of JG UNR to our UVM framework. This effort and its results are provided in Section 5.4.3.

Our functional coverage is in line with our pre-set goal of cent percent coverage. After manually justified exclusions, code coverage for the DUV attained acceptable values. Our regression suite consists of 22 test-cases each run for 100 random seeds. No failures were observed over the last several executions of the suite. Based on our coverage values, pass rate, bug rate, and bugs discovered, we conclude that we have successfully completed verification of our MESI-cache implementation using a state-of-the-art UVM test-bench.

5.2 Formal Methods

Our foremost focus during initial stages of FPV was to ensure that cover points are reached and typical behavior is observed. We went through several iterations to refine the assumptions and covers. We manually verified waveforms for typical behaviors and confirmed that the design is not trivially over-constrained. We repeated this analysis at every complexity stage, whenever a change to constraints was committed. A sample cover trace of a CPU write miss operation as seen on JasperGold is illustrated as Figure 5.3.

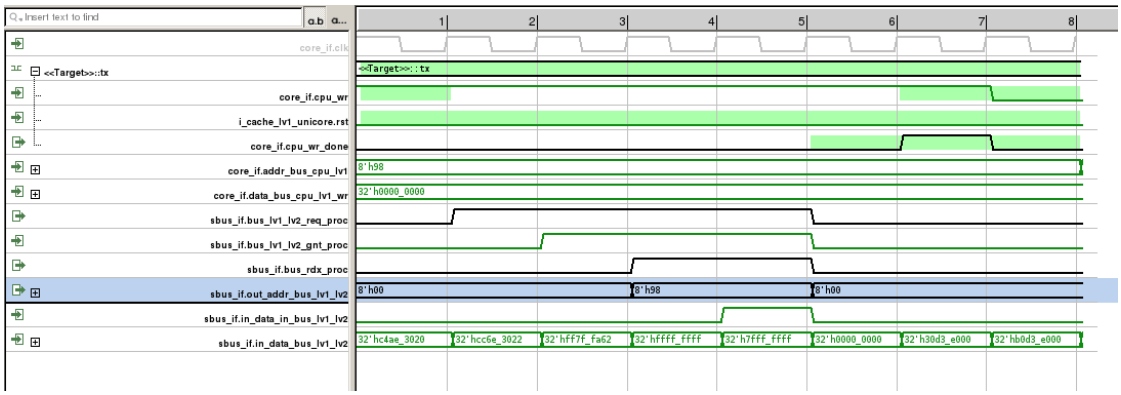

Figure 5.3: CPU write miss operation as a JG cover statement

We successfully defined assertions for all features, except the replacement algorithm, as described in the specification. A complete list of proof targets is provided in Appendix B.3. We examined properties relevant to the CPU-lv1, system bus interface, bounded liveness, MESI state transitions, coherence, memory consistency, and bug fixes. We obtained full proof for the interfaces, MESI transitions and bug fixes with reduced design parameters. We captured the definitions of cache coherence and memory consistency, namely the single-write, multiple-read invariant, and the data-value invariant, as end-to-end assertions. These invariants were explained earlier in Chapter 1. Although, we only managed a bounded proof on coherence assertions. The bound is sufficiently high for us to exude confidence about the design. We can safely say that several corner-case bugs were eliminated, and that the design is coherent and consistent.

Table 5.8: Property summary for uni-core FPV

Type	**Total**	**Proven\Reached**	**CEX\Unreachable**	**Undetermined**
Complexity Stage-1 (CS1)				
Assumes	42	N/A	N/A	N/A
Cover Statements	130	130	0	0
Assertions	55	53	0	2
Complexity Stage-2 (CS2)				
Assumes	43	N/A	N/A	N/A
Cover Statements	130	130	0	0
Assertions	56	54	0	2
Complexity Stage-3 (CS3)				
Assumes	43	N/A	N/A	N/A
Cover Statements	130	130	0	0
Assertions	56	52	0	4

As stated earlier, FPV was carried out at two levels, namely uni-core and multi-core modules, with stages of increasing complexity in each level. Summary of FPV proper-

Table 5.9: Assertion status for uni-core FPV

Type	CS1	CS2	CS3
CPU-lv1 interface	Proven	Proven	Proven
System bus interface	Proven	Proven	Proven
Liveness	Proven	Proven	Proven
MESI protocol	Proven	Proven	Proven
Bug fixes	Proven	Proven	Proven
Coherence and consistency	Bounded	Bounded	Bounded

Table 5.10: Bounded proofs in uni-core FPV

Assertion	Bound (CS1)	CS2	CS3
Liveness (CPU read)	Infinity	Infinity	Infinity
Liveness (CPU write)	Infinity	Infinity	Infinity
Coherence (CPU read)	Infinity	35	30
Coherence (LRU eviction)	45	44	41
Coherence (Snoop write-back)	Infinity	33	30
Coherence (Snoop share)	35	32	30

ties for the uni-core verification level is presented as Table 5.8. The number of assertions and assumptions change slightly between complexity stages as certain assumptions are eliminated and some assertions are clubbed together. Results of the uni-core property verification is described in Table 5.9. We observe that interface, MESI and bug fix assertions are fully proven in all complexity stages. Liveness is proven in stages CS1 and CS2, with guarantee that every CPU operation is completed within 11 cycles. This proof is possible due to simplifying assumptions (in CS1 and CS2) that access and data on the system bus is provided within a few clock cycles. In stage-3, we place realistic assumptions that the system bus access is granted in a maximum of 45 clock cycles, and that data is provided on the bus in a maximum of 9 clock cycles. These assumptions are determined by observing behavior in the multi-core FPV environment. In the worst case, when all other

level-1 caches perform eviction and write-back (each core takes 15 cycles), we obtain proc grant of shared bus in 45 cycles. Due to changes in the assumptions, liveness assertions are proven in CS3 to guarantee that CPU operations are completed in 61 clock cycles. Complete proof gives us full confidence that the uni-core model demonstrates liveness properties if access and data is provided on the system bus in a timely manner.

Bounds for the inconclusive properties in different complexity stages are listed in Table 5.10. Limits to the proof were obtained after running FPV for about 24 hours of machine time. Two of the coherence assertions are fully proven for CS1. Due to limited computing resources and time, we only manage a bounded proof for coherence properties in CS2 and CS3. We predict that a full proof can be generated with additional computing resources and time. Nevertheless, bounded proofs significantly increase confidence about the design. Cover statements in our FPV describe typical behavior in traces with a length of about 10 clock cycles. Clearly, we notice that the bound on memory consistency is of the order of 4-5 transactions. Therefore, we conclude that our analysis is sound.

Assumptions to the uni-core module, were evaluated as assertions in the multi-core instantiation environment. A summary of the properties and their results for the multi-core module are presented as Tables 5.11 and 5.12 respectively. We successfully validated assumptions to the cache module in the instantiation environment. A few of the assumptions (8/40) from uni-core stage-3, which included long sequences, only obtained a bounded proof. An example is the assumption that L1-cache obtains proc grant within 45 clock cycles after asserting system bus request. A counter-example was not found even after over 36 hours of formal analysis. Additionally, all the assumptions were verified using exhaustive simulation in the UVM regression suite. Therefore, we can state with reasonable confidence that the assumptions of uni-core FPV are valid. We also proved most uni-core assertions in the multi-core model. We obtained bounded proofs for liveness, and coherence assertions alone. A summary of bounded proofs is provided as Table 5.13. This

provides immense confidence about our proof environment.

Table 5.11: Property summary for multi-core FPV

Type	**Total**	**Proven\Reached**	**CEX\Unreachable**	**Undetermined**
Assumes	52	N/A	N/A	N/A
Cover Statements	216	216	0	0
Assertions	113	95	0	18

Table 5.12: Assertion status for multi-core FPV

Type	**Result**
Arbiter behavior	Proven
Coherence (Multi-core)	Bounded proof
Uni-core assumptions	Proven (32/40)
Uni-core assertions	
CPU-lv1 interface	Proven
System bus interface	Proven
Liveness	Bounded proof
MESI protocol	Proven
Bug fixes	Proven
Coherence (Unicore)	Bounded proof

Progress of an FPV effort is generally observed by tracking property status, assertion density, and trends in bug discovery [1]. An approximate illustration of our project progress over time is captured in Figures 5.4. Progress in FPV, as is typical, is observed in lumps towards the end of a complexity stage. A single bug fix can suddenly improve proof completion status by a large margin.

The pseudo-LRU replacement algorithm, which is excluded from FPV, is challenging to describe as properties due to the large state involved. Additionally, replacement is not

Table 5.13: Bounded proofs in multi-core FPV

Assertion	**Bound**
Liveness (CPU read)	65
Liveness (CPU write)	65
Coherence (CPU read)	28
Coherence (LRU eviction)	49
Coherence (Snoop write-back)	28
Coherence (Snoop share)	24
Multi-core Coherence (CPU read)	23

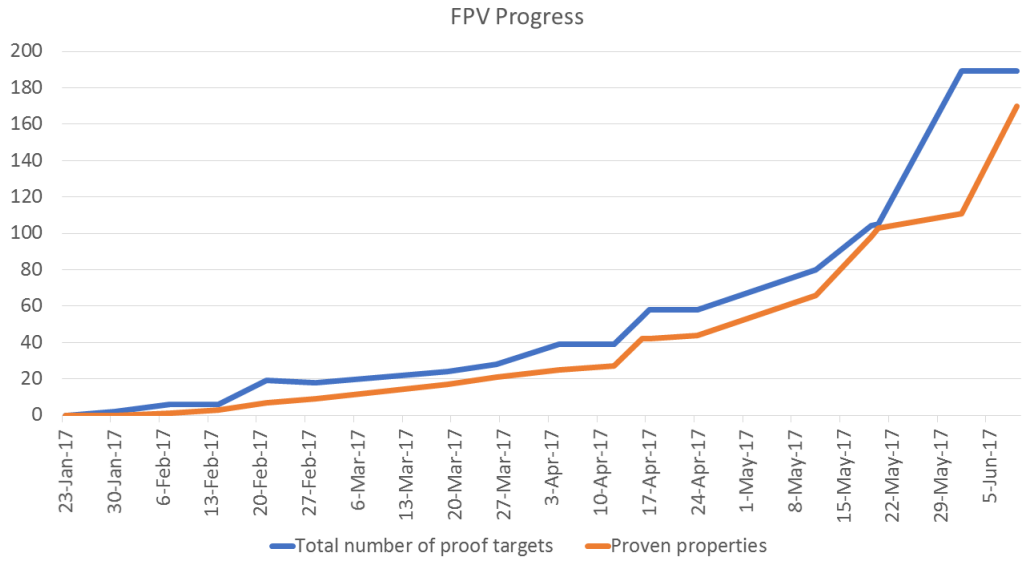

Figure 5.4: FPV progress over time

as critical as coherence, consistency and liveness. It is acceptable to lose performance due to a minor LRU bug, but the system would not function without coherence. Therefore, we prescribe LRU property verification as future work. The biggest take-away from FPV is high confidence about coherence, memory consistency and IO protocol of our cache implementation. To the best of our knowledge, this is the first effort at full-proof verification of a MESI-cache design at the RTL level, completed by leveraging parameterization.

5.2.1 Justification for Reduced Design Parameters

A prime factor, that enabled FPV of our cache at the RTL level, is design parameterization. Property verification is impossible with the original design size due to the large state space of a 256KB data cache. We proved properties by minimizing the design size. We carefully chose a set of minimal parameters that do not restrict any obvious, interesting behavior. A 7-bit address with 2-bit offset ensures that the address segregation module is utilized completely. A 2-bit set index ensures that we have 4 sets in each of the L1 caches. Each set has 4 ways and the entire data cache has 16 lines each 4 bits wide. A 3-bit address tag guaranteed that there were unique tags to trigger replacements within the set. An instruction address bound of 7'h1F ensured that interaction between instructions and data space was possible. Four instances, which is the intention of the original design, were used in the multi-core proof environment.

We understand that there is inherent risk in reducing design parameters. We could restrict interesting behavior that is possible only with the original configuration. In order to tackle this risk, we cleverly selected a reduced configuration that does not limit any obvious scenarios. Despite a careful selection of the minimal configuration, we acknowledge that parameterization can limit certain behavior in an unforeseen manner. Our intended plan was to extend our proof to the original configuration by using theorem proving techniques and induction. However, due to limited resources and time, we were not able to execute proof within a theorem proving system. Although we cannot say with certainty that the design is flawless, we have significantly high confidence about the implementation as a result of proof in the reduced setting. We defer proof with a theorem prover as future work.

5.2.2 Bugs Discovered

We identified and resolved a total of 14 design flaws using formal analysis. The highlight of our FPV effort was complete resolution of the CPU-snoop contention issue, which could not be fixed in the simulation framework. Short counter-example traces, which point to the exact source of failure, simplified analysis and verification of bug fixes.

5.2.2.1 Absence of reset signal

An obvious limitation of the legacy design, which is the lack of a global reset, was apparent immediately in FPV. Since the legacy design was not created with intent for synthesis, initial blocks were used to instantiate state elements within the design. Simulation did not raise red flags as registers were initialized either by basic CPU operations or by initial blocks. Simulation does not treat unknown or high impedance values as potential causes of failure. FPV, on the other hand, inherently performs lint checks as it requires the input model to be synthesizable. Therefore, we identified the crucial need for a global reset immediately in FPV. Although this is almost a trivial bug, it highlights the importance of RTL quality checks early in the design cycle. We resolved this bug by updating the specification to include a global reset input.

5.2.2.2 Presence of in-out ports

The legacy design made use of several in-out ports within the interface. Most signals like address, data, data_valid, bus_rd, etc. were driven by multiple entities. Although drivers of these in-out signals are mutually exclusive, these signals are often driven to high-impedance values. This can potentially lead to wrong interpretation when the design is realized in silicon. A high-impedance generally manifests as the previous active value due to capacitance. Therefore, in real life, a high-Z could incorrectly be observed as a logic high value. Once again, this relatively trivial issue brings forward the importance of

lint checks which are inherent to FPV. We resolved this bug by updating the specification to provide dedicated inputs and outputs.

5.2.2.3 Contention between CPU and snoop-side requests

The most critical bug identified is the contention issue between CPU and snoop-side requests. This issue arises as the implementation attempts to service both CPU and snoop operations at the same time. An abstract MESI model assumes that state transitions are atomic, but in reality several transitions take multiple clock cycles. In case of conflicting concurrent requests from the CPU and system bus, coherence is destroyed in our legacy implementation.

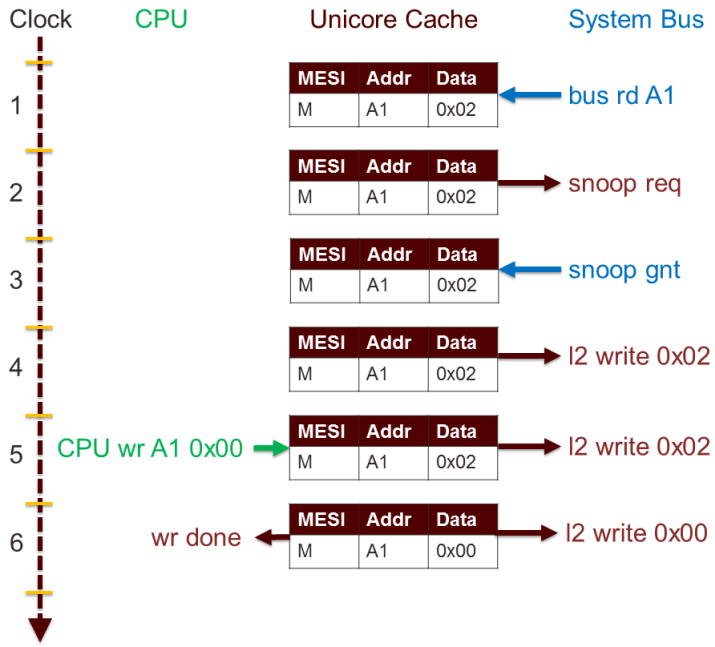

Figure 5.5: Example of contention issue

A particular example of contention is depicted in Figure 5.5. Consider a level-1 uni-

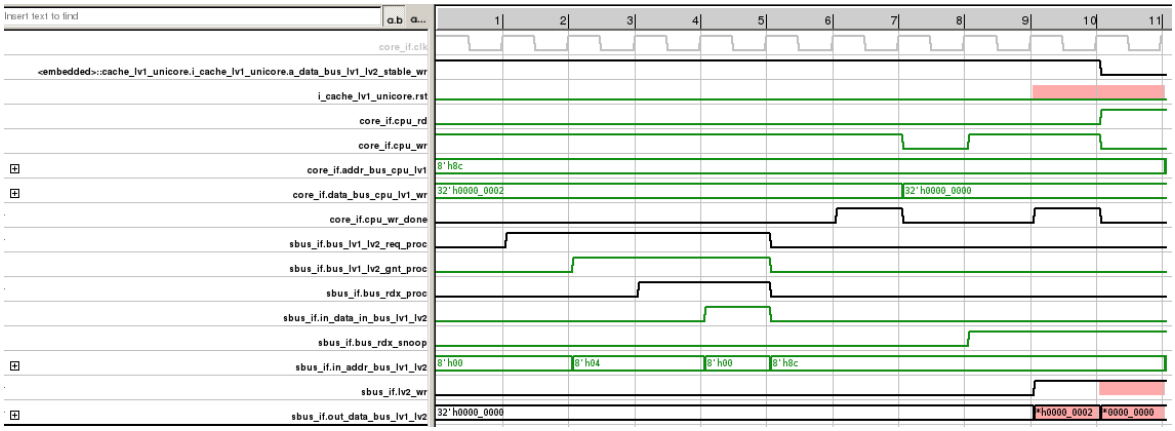

Figure 5.6: JasperGold counter-example for contention issue

core cache module with a data block A1 present in modified state. Assume that in the first clock cycle, our module receives a bus read request for A1. In order to process the bus read request, the cache module obtains snoop access to the system bus and issues a level-2 write back for A1. While the write-back is still in progress, it receives a CPU write operation for A1 in clock cycle 5. Our implementation is tuned to service both CPU and snoop in parallel. Therefore, A1 data value is updated to 0x00 and write done signal is issued to the CPU. Clearly, the concept of coherence is violated.

JasperGold generated a counter-example depicting the above scenario, early in the FPV process. An assertion defined for the system-bus interface dictated that data is unchanged as long as level-2 write signal is enabled. JasperGold provided a counter-example violating this property, to highlight the contention issue. In Figure 5.5, we notice that level-2 write-back data changes from 0x02 to 0x00 in clock cycle 6. A waveform of the trace generated in JasperGold is illustrated as Figure 5.6.

FPV demonstrated that several situations can lead to similar violations. Any two conflicting requests to the same data block can destroy coherence and consistency. We realized that in order to resolve this concurrency bug, we must prioritize either the CPU operation

or the snoop request in case of a conflict. We attempted numerous solutions with varying definitions of priority. Bugs described in Sections 5.2.2.4 and 5.2.2.5 are the results of our experiments. Ease of debug and resolution in FPV is extraordinary when compared to simulation. Often, we had no clue about the root cause of failures in simulation. It took several days to analyze long, random regression results to identify corner-case issues. On the other hand, FPV provided short traces which pointed to the exact cause of the failure. It took only a few minutes to identify the flaw in our proposed solution to the concurrency issue. Liveness properties combined with data stability assertions were most effective in debug.

Table 5.14: Solution to concurrency issue: Priority when CPU request observed first

CPU	**Bus access required**	**Snoop request**	**Bus access granted**	**Priority**
Read	No (M,E,S)	X	N/A	CPU
	Yes(I)	X	No	Snoop
		X	Yes	CPU
Write	No (M,E)	X	N/A	CPU
	Yes(S,I)	X	No	Snoop
		X	Yes	CPU

Table 5.15: Solution to concurrency issue: Priority when snoop request observed first

Snoop	**CPU operation**	**Priority**
Bus read	X	Snoop
Bus read to modify	X	Snoop
Invalidate	X	Snoop

Finally, after substantial debug, we identified and proved a working fix within the FPV environment. Our solution requires a significant change to the implementation. We define

two internal signal CPUFirst and SnoopFirst as flags generated for priority. CPUFirst indicates that in case of conflict, CPU operation should be prioritized whereas snoop request is halted. SnoopFirst indicates that snoop response must be given higher priority while CPU operation is stalled. As a corollary, both flags are mutually exclusive.

Our solution is succinctly captured in Tables 5.14 and 5.15. Note that 'X' in the tables indicates don't care. Consider the situation in which CPU operation is observed before a conflicting request on the system bus, Table 5.14 describes this scenario. If conflicting requests are observed in the same clock or snoop message is noticed first, Table 5.15 defines the solution. Essentially, snoop requests are prioritized if observed first, as captured in Table 5.15. CPU operations must be prioritized if they do not require system bus access or if our cache module has already received grant of the shared bus. SV code to realize the above solution is given below:

```
1  assign proc_req = core_if.cpu_rd | core_if.cpu_wr;
2  assign proc_resp = core_if.data_in_bus_cpu_lv1 | core_if.cpu_wr_done;
3  assign snoop_req = sbus_if.bus_rdx_snoop | sbus_if.bus_rd_snoop | sbus_if.
       invalidate_snoop;
4  assign snoop_resp = sbus_if.bus_lv1_lv2_req_snoop | sbus_if.out_data_in_bus_lv1_lv2 |
        sbus_if.invalidation_done;
5  assign conflict = (proc_req) && (snoop_req) && (`SBUS_IF_ADDR == `CORE_IF_ADDR);
6
7  assign snoop_first = snoop_req & ~cpu_first;
8
9  always@(posedge core_if.clk or posedge rst) begin
10         if(rst)
11                 cpu_first <= 1'b0;
12         else begin
13                 if (proc_resp)
14                         cpu_first <= 1'b0;
15                 else if (sbus_if.bus_lv1_lv2_gnt_proc & ~ snoop_req)
16                         cpu_first <= 1'b1;
17         end
18 end
```

Although in retrospect, our solution to the concurrency issue seems simple at the abstract level, it required significant engineering effort to be realized in RTL code.

5.2.2.4 Deadlock situation

An incorrect solution to the concurrency issue described in the Section 5.2.2.3 can easily result in deadlocks. An obvious deadlock is reached when priority flags are generated

merely on the basic of which request is observed first. Consider the case of a shared block A1 in the L1-cache. Say it receives a CPU write in the first cycle, while awaiting grant to the common bus, it receives a snoop invalidate request for block A1. Since our solution chose to prioritize CPU operation and stall the snoop request, the system fails to make any forward progress.

A complex deadlock situation is achieved when we attempted to solve the concurrency issue by generating priority flags (CPUFirst and SnoopFirst) using blk_hit_proc, an internal signal. In this solution, CPU operation is given priority if signal blk_hit_proc is generated before a conflicting snoop request is observed. Internal signal blk_hit_proc is asserted when the block requested by the CPU results in a hit (obtained from bus or previously present in cache). This fix works for most scenarios except when both blk_hit_proc and a conflicting snoop request arrive in the same clock cycle. If we chose to stall the snoop request in this scenario, deadlock is achieved for specific CPU requests.

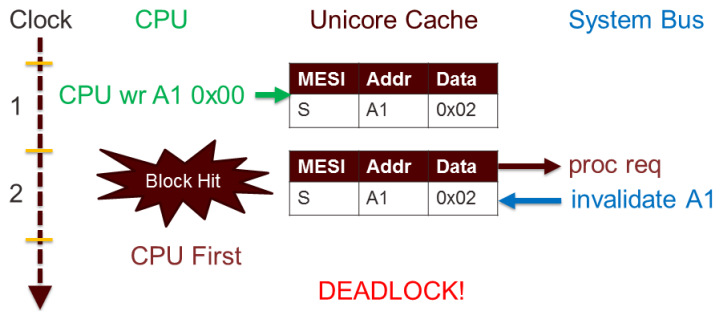

Figure 5.7: Example of deadlock

An example of this complex deadlock is illustrated in Figure 5.7. Consider a data block A1, present in shared state. CPU issues a write operation on A1 in the first clock cycle. System bus access is requested on the next clock, simultaneously an invalidate request for A1 is received. Signal blk_hit_proc is asserted at the same time as the incoming

invalidate. Therefore, based on our solution, we halt processing of the snoop request and prioritize the CPU write which resulted in a cache hit. Snoop request cannot proceed without invalidation. CPU write will not complete before system bus access. Hence, deadlock is achieved.

This complex deadlock showed up only after several days of regression analysis in the simulation framework. FPV, on the other hand, highlighted the issue almost immediately. Interestingly a plausible deadlock situation was discovered in FPV even after simulation-based verification was assumed to be complete. Within the implementation, we forgot to raise bus request even if CPU operation was stalled due to a conflict. Although extremely improbable, this could lead to an endless chain in which snoop requests are continuously given higher priority. We identified this issue in FPV and fixed generation of the bus request signal.

5.2.2.5 Livelock situation

A livelock is a scenario in which there is constant activity in the system, but it fails to make any overall progress. Consider the solution based on internal signal blk_hit_proc as described in Section 5.2.2.4. The only change is that if snoop and block hit rise on the same clock edge, snoop transactions are given priority while CPU operations are stalled. An example of how this solution leads to deadlock is illustrated as Figure 5.8.

Consider a write request to data block A1 in the first clock cycle. Our uni-core cache obtains bus access and receives A1 in exclusive state. In the same cycle that block hit is generated, a bus_rdx request to A1 is observed. Since our policy dictates that CPU operation is halted in such a scenario, we invalidate A1 and service the snoop request. As the CPU write is still incomplete, we re-issue the bus request. Our cache module obtains bus grant, and immediately generates bus_rdx request for A1. In such a manner, bus grant and A1 is constantly exchanged between two requesting L1-caches, resulting in a livelock.

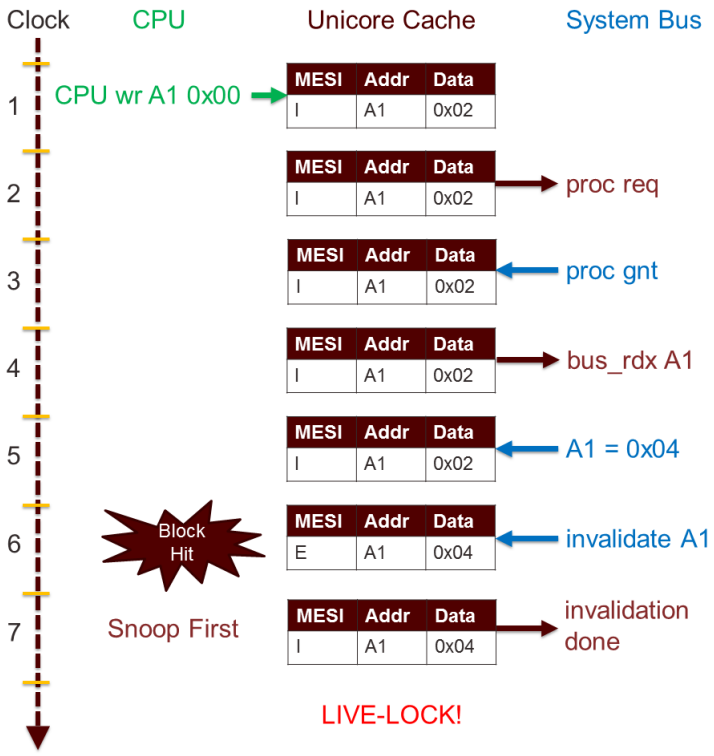

Figure 5.8: Example of livelock

Ultimately, we avoid deadlocks and livelocks by defining priority flags based on outputs from the arbiter. Arbiter is the ultimate authority on resolution of conflicts in our system. It is responsible for serializing operations, in effect defining the correct sequential order. Our clean solution to the concurrency issue was described in Section 5.2.2.3.

5.2.2.6 Bus request without CPU operation

A cousin bug of the concurrency issue is that bus request is asserted even without an active CPU operation. A counter-example from JasperGold is provided as Figure 5.9. A conflicting snoop operation invalidates the cache block exactly in the same cycle in which the CPU operation is serviced. As a result, there is a spurious bus request on the next cycle. This bug is resolved with the fix for the concurrency issue.

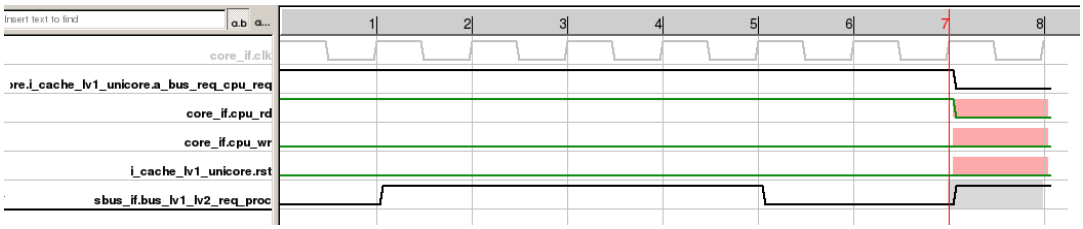

Figure 5.9: Counter-example highlighting bus request without CPU operation

5.2.2.7 *Signal cp_in_cache for incoming invalidate request*

As a result of the concurrency fix, whenever a snoop request was halted due to conflict and CPU first priority, signal cp_in_cache was asserted irrespective of the snoop request. However, the specification dictates that cp_in_cache should be generated for bus_rd or bus_rdx requests only. We observed incorrect behavior as a counter-example to one of the system bus assertions.

5.2.2.8 *Invalidation acknowledgment during CPU first priority*

Initially, acknowledgment for an invalidate request is asserted immediately if the block is not present in cache. We assumed that this behavior would not be affected by the concurrency fix. However, we later realized through FPV that invalidation done signal should be halted if there is a conflict. This is another side effect bug of the incomplete concurrency fix. In this case, priority flags are generated correctly, but they are not fully utilized to prevent incorrect behavior.

5.2.2.9 *De-assertion of CPU first priority*

The concurrency fix suggested in Section 5.2.2.3, dictates that priority flag CPUFirst is de-asserted when a response is provided to the CPU. Our implementation was made in the data cache functionality. We incorrectly ignored a response from instruction cache. Consider the following situation: an instruction read obtains bus grant, which asserts the

CPUFirst signal, but completion of the read request does not de-assert CPUFirst. As a result, the next data operation is incorrectly given higher priority than an incoming snoop request, which leads to severe failures. This error is fixed by taking into consideration response from the instruction cache as well.

5.2.2.10 Incorrect next-state logic in MESI FSM

Combinational logic to determine next-state function, within the MESI finite state machine, was realized incorrectly in the legacy design. It made the assumption that a snoop-side invalidate request is impossible for a cache block existing in modified/exclusive state. Uni-core FPV pointed out the existence of this flaw. We resolved this issue by supporting invalidate requests for modified and shared data blocks.

5.2.2.11 LRU state implemented as latches

The LRU state for each cache set was implemented as a latch. Latches can cause issues in a design based on a clock. FPV, which also performs a lint check on the RTL, pointed out the presence of latch inference, early in the verification process. We resolved this bug by recoding the RTL.

5.2.2.12 De-assertion of signal data_in_bus_cpu_lv1

Level-1 cache response to a CPU read operation is through the signal data_in_bus_cpu_lv1. In the legacy design, this output remains high for one additional clock cycle after CPU read request is de-asserted. As a result, in case of consecutive read and write requests, output data_in_bus_cpu_lv1 is high when a CPU write operation is in progress. This is undesirable and can lead to confusion in the core. We eliminate this possibility by a logical AND operation with CPU read request.

5.2.2.13 De-assertion of signal cpu_wr_done

Level-1 cache responds to a CPU write operation with cpu_wr_done to signal that the write is successfully completed. Similar to the previous bug, cpu_wr_done remained high during a CPU read operation. This is avoided by simply performing a logical AND with the write request.

5.2.2.14 Multiple drivers for lv2_wr_done

Multi-core FPV identified the presence of multiple drivers for the lv2_wr_done signal. In the implementation, this port is declared as an output of the multiplexer block within the multi-core module. However, it is also driven by independent glue logic within the module. This is another example of a lint error spotted by JasperGold.

5.2.3 Coverage

FPV tools including JapserGold provide coverage metrics to track the percentage of logic covered as properties. Although these measures are not as robust as coverage in the simulation framework, it provides a rough idea of verification progress. Coverage in FPV is primarily based on the logic that falls within the cone of influence of defined assertions. Occasionally, additional logic might be considered part of an assertion's logic cone even without a significantly direct impact. Therefore, these metrics should be treated as a rough estimate rather than the sign-off criteria. We provide our line and signal coverage numbers in Table 5.16. Since we haven't defined assertions for LRU functionality, a slight dip in coverage is expected.

Table 5.16: Coverage metrics for FPV

Type	**Line Coverage**	**Signal Coverage**
Uni-core module	97.08%	88.42%
Multi-core environment	96.65%	73.10%

5.2.4 Formal as Design Aid

Formal techniques are handy tools in the hands of design engineers as well. We encountered numerous applications of formal analysis specific to RTL design during our verification process. We outline these usage scenarios in this section. Table 5.17 provides details of our minimized configuration, to help appreciate the design size and complexity feasible for formal analysis. Our FPV exercise demonstrates that formal methods are effective for complex, real-life designs with suitable parameterization.

Table 5.17: Design size for our FPV

Type	**PI**	**PO**	**State elements**	**Property flop bits**
Uni-core module	39	29	1117	748
Multi-core environment	68	46	2926	1228

Scenarios for the use of formal technology as design aid, exercised in our study, are listed below:

1. *Automatic lint check*

   FPV tools including JasperGold inherently perform a lint check of the RTL for synthesis. With reference to our study, FPV highlighted quality issues like the absence of reset, presence of in-out ports, and latch inference. Usually these issues are identified by a lint tool later during the synthesis stage. A quality lint check in the form of FPV, at an early design stage is helpful. JasperGold provides a specific application called Automatic Formal Lint (AFL), which specializes in RTL quality checks.

2. *Legacy design understanding*

   Design implementations are usually passed through teams and engineers across several years. Ownership is often transferred to new designers, who have no clue about

the previous design. Sufficient documentation to understand the implementation is often absent. This is similar to how we obtained a legacy implementation of the cache controller. Formal property verification is extremely useful in such scenarios. Cover statements and simple properties serve as an instant test-bench. We are able to generate interesting behavior with minimal effort. A prime advantage in FPV is that we are required to merely specify the destination, the tool figures out the shortest path to achieve that state. This is in stark contrast from simulation, where every step of the journey must be explicitly defined. With reference to our project, say we wish to observe behavior when a write operation is performed on a shared block, we simply specify a cover statement.

3. *Design exercise FPV*

   Designers can use FPV to bring up an instant test-bench with minimal effort. Typical behavior can be analyzed without spending significant time on the development of UVM components. It is acceptable to over-constrain the design in such an exercise, as robust verification will be performed later. With simplified assumptions, early design flaws can be weeded out before release to the verification team.

4. *Validation of bug fixes*

   Major design updates, like the concurrency fix in our project, often lead to numerous additional bugs as a side effect. If we can define the micro-architectural update as properties, a clean implementation can be obtained with few iterations in FPV. We come to this conclusion based on our experience with the concurrency fix. Additional holes within the implementation were identified immediately with assertions that represented the update at an abstract level.

5. *Formal equivalence verification for RTL*

   FEV is a powerful technique to guarantee that functionality is preserved with design

changes like parameterization, de-feature bits, clock gating, and timing fixes. FEV is often faster than performing a complete FPV execution on the new design. When such changes are made late in the design process, FEV can quickly provide confidence that functionality is preserved. We utilized FEV to verify parameterization of the number of cores. We performed combinational equivalence between the old implementation and the parameterized model with default configuration.

6. *Performance verification*

   Property verification can guarantee performance bounds for a given design if complexity is manageable. In our study, we fixed the latency for liveness properties. Under specific assumptions on level-2 performance, we could ensure that a CPU operation is serviced within a fixed number of clock cycles. We first start with the least latency value, and analyze the counter-example to evaluate if behavior is valid. In complexity stage-2 of our uni-core FPV, we proved that both CPU read and write is serviced within 11 clock cycles, based on the assumption that shared bus access is granted immediately. If the scenario is acceptable, we increase the latency bound in the property. Otherwise, we fix the RTL to ensure performance requirement is met.

5.3 Comparison between Formal and Simulation

In this section, we offer an elaborate comparison between our FPV and simulation efforts. We comment about the ease of debug, bug discovery, trace length, required expertise and corner case errors in each approach. It is fairly clear that both technologies have their own advantages and limitations. Neither formal not simulation can fully replace the alternative approach, at least with the current tool capabilities.

Table 5.18 briefly summarizes a general analysis of simulation and formal techniques. The 3 pillars which form the focus of simulation-based verification are stimulus, checkers and coverage. Formal analysis is centered around defining properties and controlling state

Table 5.18: Generic analysis of formal vs simulation

	Simulation framework	**Formal verification**
Focus	Random stimulus Functional coverage Checkers	Properties State space exploration
Challenges	Corner cases Time to build infrastructure Debug	Data transformation blocks Property definition Complexity (state space)
Typical trace length	Large	Short
Ease of debug	Medium	Easy
Expertise required	Low	Medium
Time to first bug	2-3 weeks	1-2 days
Complex bugs	Hit after several regressions	Easy analysis of concurrency, deadlock , and livelock

space. Therefore, both techniques require a fundamentally different perspective. Simulation excels in large designs like system-on-a-chip(SoC) implementation. SoC level verification is impossible using formal techniques due to the immense state space problem. Nevertheless, simulation is deficient in the analysis of corner cases, it fails to exhaustively verify the design. Formal techniques are suited to rigorously verify control blocks and data transport modules, by highlighting subtle corner case issues [40]. It is challenging to formally verify data transformation blocks which operate on huge chunks of data, due to complexity. Formal analysis is challenging to apply in verification of packet-based protocols. But with techniques like parameterized hardware design, we believe that formal verification of large packet based protocols can be made feasible.

Within the simulation framework, verification cannot begin until the necessary infrastructure is substantially developed [24]. This includes checkers, drivers, monitors and coverage collectors. As a result, it takes several weeks even for a basic investigation of the design. Simulation traces are generally long, making debug challenging as compared to formal. It is common for an error to show up several thousand cycles after the primary

source of bug. Engineers manually trace back several transactions to identify plausible causes of failure. Additionally, complex issues like concurrency, deadlocks and livelocks are hard to reason about in simulation [41]. Identification of complex issues relies on regressions with pseudo-random stimulus. Despite the drawbacks, simulation is the de-facto standard in the industry due to the limited expertise requirement.

Formal methods, on the other hand, are capable of creating instant test-benches. We can perform verification of typical behavior as soon as the design is available. Therefore, formal techniques can identify erroneous behavior almost immediately. It is easy to understand and debug complex issues like concurrency and deadlock. This is evident in our study, contention between the CPU and snoop was identified in simulation, but debug and resolution was impossible due to long random traces. Analysis of concurrency was extremely simple within FPV as it provided short traces pin-pointing the exact cause of failure. Therefore, debug is significantly simpler in formal. Nevertheless, engineers require additional training and expertise before successful application of formal methods.

Seligman et al. assert that formal verification provides a greater return on investment (ROI) as compared to simulation in terms of engineering and machine costs [1]. An increase in ROI of about 2-9x was observed for formal analysis of real designs at Intel. We compare results from the UVM framework and FPV in Table 5.19. Approximately ten weeks of focused engineering effort was spent on simulation, while about 8 weeks were dedicated to FPV. Our study is inherently disadvantageous to FPV as the formal effort succeeded the simulation exercise. Formal verification provides maximum benefits when introduced early in the design process [1]. Although earlier FV was suitable only with mature RTL due to the long analysis time, FV tools in their recent capacity provide maximum benefits early in the design process. In spite of the clear disadvantage, we observe that FPV provides better ROI in all aspects of comparison. The number of bugs discovered in either approach is comparable. However, the quality of bugs identified in FPV is

Table 5.19: Comparison of results from formal vs simulation

	UVM	**FPV**	**FV advantage**
Engineering effort	10 weeks	8 weeks	-
Bugs discovered	13	14	1.07x
Bugs (%)	48%	52%	-
Bugs/Eng effort	1.3	1.75	1.35x
Simple bugs	8	7	-
Complex bugs	5	4	-
FV quality bugs	0	3	-
Scaled bugs	8*1 + 5*2 = 18	7*1 + 4*2 + 3*4 = 27	1.5x
Scaled bugs/Eng effort	1.8	3.375	1.875x
Time to deploy	21 days	2 days	10.5x
Typical trace length	40000 cycles	6-40 cycles	100x
Typical execution time	3 hours	1 hour	3x
Time to concurrency issue	6 weeks	2 weeks	3x
Debug of complex bugs	1-3 days	1-2 hours	24x

significantly higher. If we assign weights based on complexity, FPV ROI is about 1.875x that of simulation in terms of engineering effort. The significance of FPV is supported by the fact that resolution of the concurrency issue was infeasible in simulation. The biggest advantages of FPV are ease of debug and short counter-examples. They enabled implementation of an easy, clean fix for the contention bug. Debug time is reduced by about 20x while trace length is shortened by a factor of hundred.

Despite its obvious advantages, formal analysis is limited by design complexity and quality of properties. A feature of the DUV, which is not captured as a property cannot be verified in FV. Often, it is challenging to describe certain features as properties without adding significant state space. In our study, we excluded verification of LRU functionality in FPV due to the number of state elements required. FPV can only verify behavior that is explicitly specified. Contrastingly, pseudo-random simulation can fortuitously identify faulty behavior which is not explicitly specified as well.

5.4 Collaboration between Formal and Simulation

It is fairly obvious that neither formal nor simulation can completely replace the alternative approach. Both techniques have their own unique advantages and deficiencies. During the course of our study, we identified opportunities for successful collaboration between formal and simulation. We believe that both approaches can co-exist to supplement each other, in a productive manner without compromising engineering effort. There are several opportunities for a mutually beneficial arrangement. In this section, we describe specific events in our study where we leveraged formal techniques within simulation and vice-versa. We implore the use of formal methods as an integral part of the design process. It essentially serves as a new form of interacting with the RTL.

5.4.1 Validating FPV Assumptions in Simulation

Assumptions form the basis of proof in a formal environment. It is crucial to verify the set of constraints used to generate proof. This is achieved in three methods, namely arguments supported by evidence in the specification, proving assumptions in the instantiation environment, and extensive simulation. In our study, we attempted to prove assumptions in the multi-core environment in addition to validating them in simulation. We observed that it is relatively easier to verify assumptions in a extensive regression suite. All assumptions used in uni-core FPV passed as assertions in our UVM regression suite. Proof in the instantiation environment is not always feasible due to complexity of the environment. In such a situation, simulation-based verification at a higher level of hierarchy can be used to gain confidence about assumptions used in FPV at the module level.

5.4.2 Bug-hunting FPV

In case of a risky design feature, bug hunting FPV can be performed on the critical function in addition to simulation-based verification on the entire module. Bug hunting

exercise is simpler compared to full-proof verification. Complexity is easier to handle as properties are focused only on a portion of the logic. Additionally, we could black-box irrelevant parts of the design. In our study, a bug hunting exercise centered around coherence would yield good results when performed in addition to the UVM framework. Our full-proof verification effort, which subsumes a bug hunting exercise, discovered 12 bugs not previously identified in the UVM environment.

5.4.3 Improving Simulation Code Coverage

Hundred percent code coverage is a significant sign-off criteria for the design. Engineers spend weeks analyzing uncovered portions of the RTL code. Formal analysis can significantly speed up this process by determining unreachable portions of RTL [1]. JapserGold provides a application called UNR for this purpose. Our raw code coverage metrics from simulation were presented earlier in Table 5.5. UNR provides a list of unreachable points, which we can categorize as either waive or fix. Greater confidence is achieved when items are excluded after formal analysis has determined un-reachability. We witnessed an improvement of about 0.5% to 2% without any significant effort on JG UNR. These unreachable points were determined without any initialization or environmental constraints. With sufficient input constraints, we believe that significant coverage holes can be identified easily using JG UNR.

5.4.4 Additional Opportunities

A non-exhaustive list of additional possibilities is given below:

- Connectivity verification at SoC level [33]
- Control register verification
- Post-silicon debug [34]
- Security verification using information flow tracking [42]

6. CONCLUSIONS

In summary, we successfully utilized formal property verification to gain immense confidence about our MESI-based cache implementation in SystemVerilog. We demonstrate that formal verification of cache controllers is feasible at the RTL level with suitable parameterization. We performed full-proof verification of features, except LRU operation, for our design in a reduced configuration. Although the minimized configuration does not explicitly restrict any interesting behavior, we defer a formal proof of the original size design using theorem proving techniques as future work. We successfully defined interface-level assertions for high level properties like coherence and memory consistency. Two invariants, which constitute coherence in a shared memory system, namely the SWMR and DV invariants are described as end-to-end assertions in SystemVerilog. Additionally, we verified functionality of the communication infrastructure and protocol adherence of the input-output interface using SystemVerilog assertions.

We provide an elaborate comparison between a state-of-the-art UVM environment and FPV. We utilized de-facto industry approaches like pseudo-random stimulus, metric-driven verification, and IP methodology, to develop a rigorous, parameterized, simulation framework. We identified a total of 13 bugs using the UVM test-bench. However, it was extremely challenging to reason about complex issues like concurrency, deadlock and live-lock. Large failure traces, with a run-time of several thousand cycles, rendered debug of complex issues inefficient and infeasible. Corner case errors were identified predominantly using random regressions, which required time-consuming analysis in order to identify the source of failure.

Formal property verification identified a total of 14 design flaws, out of which 3 are FV quality bugs that would be improbable to identify in random regressions. Despite the

fact that FPV succeeded the UVM effort, FPV recognized a comparable number of design flaws, unresolved in the simulation environment. The three FV quality bugs are described in Sections 5.2.2.4, 5.2.2.5, and 5.2.2.8. We identified and resolved a critical concurrency issue with the potential to destroy notions of coherence and consistency in our system. This complex issue was the result of simultaneous, conflicting CPU and snoop-side requests. Issues of deadlock and live-lock, which arose as a result of attempted fixes to the concurrency bug, were resolved in FPV with minimal effort. In line with previous observations made by Seligman et al. [1], we note that FV offers a higher return on investment in terms of engineering effort. A quick comparison of results, yields an FPV advantage of 1.875x. FPV using parameterization enabled quick convergence of our design implementation. We conclude that it is substantially simpler to understand and resolve complex issues like concurrency in a formal environment.

Formal and simulation are often incorrectly viewed as disparate, independent technologies. We note that there are numerous opportunities for successful collaboration between the two approaches. Formal methods can complement simulation with specialized application in connectivity, bug-hunting, register verification, linting, coverage improvement and security among numerous possibilities. In our study, we demonstrated the use of formal as coverage improvement and bug-hunting tools. Alternatively, the assumptions in a formal environment can be verified using extensive simulation at a higher level of design hierarchy. Training designers and regular verification engineers in ABV is immensely beneficial for the design process. FV is useful throughout the design process, popular use cases include: design-exercise FPV, bug-hunting FPV, post-silicon debug, connectivity verification, clock gating validation, RTL-netlist FEV, control register verification, and coverage improvement. However, they are not limited to these specific usage models at particular stages of the design cycle. Formal techniques serve as a generic tool-kit rather offering new methods to interact with the RTL.

We offer the following generic guidelines, based on our experience with formal verification, to maximize productivity in digital design.

1. *Apply FV early*

   FV methods provide significant advantages when applied early in the design cycle. Lightweight usage models like design-exercise FPV can even apply to large and complex designs by focusing on specific behavior. FV provides an instant test-bench to observe typical behavior without significant effort.

2. *Parameterize the implementation*

   Identify opportunities to parameterize the design at every stage. Parameterization can reduce serious complexity making it feasible for FV.

3. *Design as components tangible for FV*

   This is part of a greater envisioned plan for formally verified libraries. If modules are designed in sizes that permit FV, components can be rigorously verified using FV methods while simulation could serve the verification need at a higher level.

4. *Specific application of FV*

   In cases where full scale application is infeasible, we suggest identifying specific opportunities where Fv can provide better returns. Particular examples are connectivity, control register, security verification, post-silicon debug and coverage improvement.

5. *Bug-hunting FPV*

   In case of a critical feature, bug-hunting FPV can be performed to improve confidence about the design.

6. *FEV for design changes*

   FEV is useful in verifying design updates like parameterization, clock gating, and

timing fixes. FEV effort in such cases is more robust when compared to simulation regressions. It is often easier than a full-fledged FPV exercise as well.

6.1 Future Work

We enumerate opportunities to extend current work below:

1. *Formal proof of coherence in original design configuration*

   In this study, we gained confidence of coherence and a sound communication infrastructure for a minimized configuration of the design. It would be interesting to extend our proof to the general setting with theorem proving techniques.

2. *Transactional equivalence between reference model and RTL*

   In our UVM environment, we developed a transaction-accurate reference model of our design. An equivalence check between the transaction-level model and RTL would evaluate the applicability of FEV tools for such a use case.

3. *Formal verification of complex caches*

   Our study demonstrated the feasibility of FPV for a base-line MESI implementation with a blocking, shared bus. Modern cache implementations are aggressively optimized for performance with features like split-transactions, and write-buffers. The communication infrastructure in a shared memory system is constantly evolving. Bus architecture is replaced with NoC topologies like 2D mesh, or torus. Additionally, recent concepts like hardware transactional memory further introduce complexity to caches [43]. We suggest extension of our analysis to more complex designs.

4. *Security verification of caches*

   Cache side-channel and timing attacks severely compromise hardware security. Cryptographic algorithms like AES and RSA were violated by an unprivileged user program based on timing information from cache misses [44]. We propose verification

of security aware caches including partition-locked cache (PL cache) using formal techniques. Specifically, SecVerilog enables security verification at the hardware level by annotating SystemVerilog code with information flow tracking labels [42]. Analysis of formal approaches for security verification can further strengthen the practical usage portfolio of FV.

REFERENCES

[1] E. Seligman, T. Schubert, and M. V. A. K. Kumar, *Formal Verification: An Essential Toolkit for Modern VLSI Design*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2015.

[2] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st ed., 2011.

[3] M. M. K. Martin, "Formal verification and its impact on the snooping versus directory protocol debate," in *2005 International Conference on Computer Design*, pp. 543–549, Oct 2005.

[4] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/-Software Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

[5] N. Dave, M. C. Ng, and Arvind, "Automatic synthesis of cache-coherence protocol processors using bluespec," in *Proceedings of the 2Nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '05, (Washington, DC, USA), pp. 25–34, IEEE Computer Society, 2005.

[6] D. Vantrease, M. H. Lipasti, and N. Binkert, "Atomic coherence: Leveraging nanophotonics to build race-free cache coherence protocols," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pp. 132–143, Feb 2011.

[7] V. Nagarajan, "Lecture 5: Snooping coherence protocol," 2017.

[8] Intel, *Intel Core2 Extreme Processor X6800 and Intel Core2 Duo Desktop Processor E6000 and E4000 Sequence*.

[9] F. Verbeek, P. M. Yaghini, A. Eghbal, and N. Bagherzadeh, "Deadlock verification of cache coherence protocols and communication fabrics," *IEEE Transactions on Computers*, vol. 66, pp. 272–284, Feb 2017.

[10] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness, "Verification of the futurebus+ cache coherence protocol," *Formal Methods in System Design*, vol. 6, no. 2, pp. 217–232, 1995.

[11] F. Pong, M. Browne, A. Nowatzyk, and M. Dubois, "Design verification of the s3.mp cache-coherent shared-memory system," *IEEE Transactions on Computers*, vol. 47, pp. 135–140, Jan 1998.

[12] F. Pong and M. Dubois, "Verification techniques for cache coherence protocols," *ACM Comput. Surv.*, vol. 29, pp. 82–126, Mar. 1997.

[13] G. Delzanno, *Automatic Verification of Parameterized Cache Coherence Protocols*, pp. 53–68. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000.

[14] C.-T. Chou, P. K. Mannava, and S. Park, *A Simple Method for Parameterized Verification of Cache Coherence Protocols*, pp. 382–398. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[15] J. Bingham, J. Erickson, G. Singh, and F. Andersen, "Industrial strength refinement checking," in *2009 Formal Methods in Computer-Aided Design*, pp. 180–183, Nov 2009.

[16] E. A. Emerson and V. Kahlon, *Exact and Efficient Verification of Parameterized Cache Coherence Protocols*, pp. 247–262. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.

[17] Y. Li, K. Duan, Y. Lv, J. Pang, and S. Cai, "A novel approach to parameterized verification of cache coherence protocols," in *2016 IEEE 34th International Conference*

on Computer Design (ICCD), pp. 560–567, Oct 2016.

[18] R. Komuravelli, "Verification and performance of the denovo cache coherence protocol," 2010. Master's thesis at the University of Illinois at Urbana Champaign.

[19] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "Denovo: Rethinking the memory hierarchy for disciplined parallelism," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, (Washington, DC, USA), pp. 155–166, IEEE Computer Society, 2011.

[20] J. F. Cantin, M. H. Lipasti, and J. E. Smith, *Dynamic Verification of Cache Coherence Protocols*, pp. 25–42. New York, NY: Springer New York, 2004.

[21] A. DeOrio, A. Bauserman, and V. Bertacco, "Post-silicon verification for cache coherence," in *2008 IEEE International Conference on Computer Design*, pp. 348–355, Oct 2008.

[22] A. DeOrio, I. Wagner, and V. Bertacco, "Dacota: Post-silicon validation of the memory subsystem in multi-core designs," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pp. 405–416, Feb 2009.

[23] Cadence, "Modeling and verifying cache-coherent protocols, vip, and designs." White paper by Cadence.

[24] C. Spear, *SystemVerilog for Verification, Second Edition: A Guide to Learning the Testbench Language Features*. Springer Publishing Company, Incorporated, 2nd ed., 2008.

[25] R. Salemi, *The UVM primer*. Boston Light Press, 2013.

[26] H. Zhaohui, A. Pierres, H. Shiqing, C. Fang, P. Royannez, E. P. See, and Y. L. Hoon, "Practical and efficient soc verification flow by reusing ip testcase and testbench," in *2012 International SoC Design Conference (ISOCC)*, pp. 175–178, Nov 2012.

[27] S. Vijayaraghavan and M. Ramanathan, *A practical guide for SystemVerilog assertions*. Springer Science & Business Media, 2005.

[28] M. Graphics, "Verification academy," *UVM Cookbook. Mentor Graphics*, pp. 1–569, 2012.

[29] Cadence, "SystemVerilog advanced verification using UVM," 2012.

[30] U. V. M. Accellera, "1.1 users guide," 2011.

[31] E. Cerny, S. Dudani, J. Havlicek, and D. Korchemny, *SVA: The Power of Assertions in SystemVerilog*. Springer Publishing Company, Incorporated, 2nd ed., 2014.

[32] R. K. Ranjan, C. Coelho, and S. Skalberg, "Beyond verification: Leveraging formal for debugging," in *2009 46th ACM/IEEE Design Automation Conference*, pp. 648–651, July 2009.

[33] S. K. Roy, "Top level soc interconnectivity verification using formal techniques," in *2007 Eighth International Workshop on Microprocessor Test and Verification*, pp. 63–70, Dec 2007.

[34] S. Ray and W. A. Hunt, "Connecting pre-silicon and post-silicon verification," in *2009 Formal Methods in Computer-Aided Design*, pp. 160–163, Nov 2009.

[35] V. Kottapalli and F. Andersen, "Formal property verification of a MESI-based cache implementation," 2017. Poster presented at IEEE Texas Workshop on Integrated System Exploration.

[36] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking," in *Proceedings of the 2006 IEEE/ACM Interna-*

tional Conference on Computer-aided Design, ICCAD '06, (New York, NY, USA), pp. 836–843, ACM, 2006.

[37] A. Koelbl, Y. Lu, and A. Mathur, "Embedded tutorial: formal equivalence checking between system-level models and rtl," in *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.*, pp. 965–971, Nov 2005.

[38] P. Chauhan, D. Goyal, G. Hasteer, A. Mathur, and N. Sharma, "Non-cycle-accurate sequential equivalence checking," in *2009 46th ACM/IEEE Design Automation Conference*, pp. 460–465, July 2009.

[39] M. V. A. K. Kumar, A. Gupta, and S. S. Bindumadhava, "RTL2RTL formal equivalence: Boosting the design confidence," in *Proceedings 2nd French Singaporean Workshop on Formal Methods and Applications, FSFMA 2014, Singapore, 13th May 2014.*, pp. 29–44, 2014.

[40] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik, *Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation*, pp. 414–429. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[41] S. Tasiran, Y. Yu, and B. Batson, "Linking simulation with formal verification at a higher level," *IEEE Design Test of Computers*, vol. 21, pp. 472–482, Nov 2004.

[42] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, (New York, NY, USA), pp. 503–516, ACM, 2015.

[43] V. Kottapalli and S. Khatri, "A practical methodology to validate the statistical behavior of bloom filters," in *2016 International Conference on Hardware/Software*

Codesign and System Synthesis (CODES+ISSS), pp. 1–8, Oct 2016.

[44] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, (New York, NY, USA), pp. 494–505, ACM, 2007.

APPENDIX A

SPECIFICATION

This chapter describes the high-level specification of our design. The environment consists of a configurable N-core system with private L-1 cache for each core, an atomic snooping bus for communication between caches, an arbiter to determine bus access, and communication between LLC and main memory. A block diagram of the system is illustrated in Figure A.1. The design (DUV) encompasses the multi-core L1 cache alone. However, we include the level-2 cache and arbiter behavior for completeness. A key aspect to remember is that the specification provided here is the latest version. We began with a simple toy specification, which evolved into a full-fledged architectural definition through the design and verification process. Several changes to the implementation like addition of reset, removal of input-output ports, performance upgrades, etc resulted in refinements to the specification as well.

We focus primarily on the functional aspects of the L1 cache, and avoid severe performance optimizations. Operations on the CPU-L1 interface are blocking, implying that the core must keep the request asserted until it gets a response from the cache. Similarly, the common system bus is also atomic and blocking. As a result, at any instant, a maximum of one request is pending on the system bus.

Salient features of our system are:

- Parameterized implementation
- Private L1 cache and shared L2 cache
- Separate data and instruction cache in level-1; Unified level-2 cache

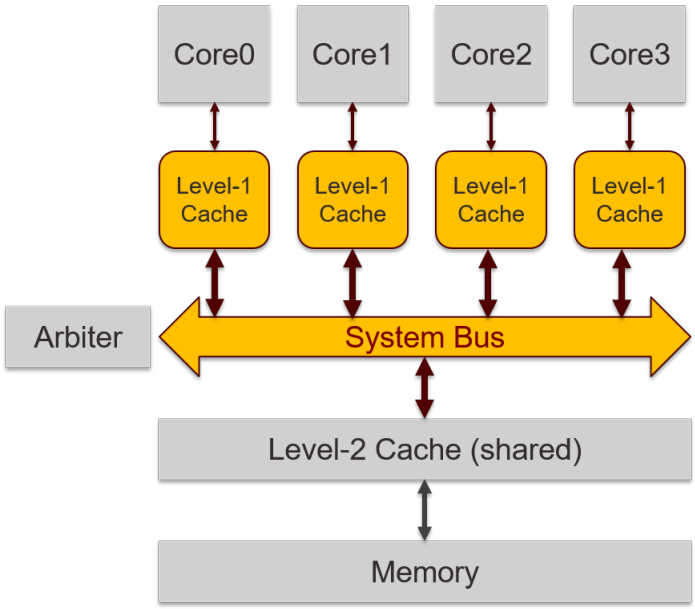

Figure A.1: Block diagram of the complete system

- 4-way associative L1 cache and 8-way L2 cache
- Physically indexed physically tagged (PIPT) cache system; No translation look-aside buffer (TLB) needed
- Inclusive cache
- MESI-based coherence protocol
- Pseudo-LRU replacement policy
- Write-back and write-allocate schemes
- No write buffers

Our implementation is parameterized to support re-use, modularity, and formal analysis. A list of configurable elements is provided below:

1. **Data width:** determines the data bus width, and size of each cache block

2. **Address width:** determines the address space and address bus width

3. **Number of index bits:** defines the number of sets within the cache, subsequently controlling size of the data and instruction L1 cache

4. **Tag width:** determines address tag bits for each cache line

5. **Number of cores:** controls the number of CPUs within the system

6. **Instruction address bound:** defines division of address space between data and instructions

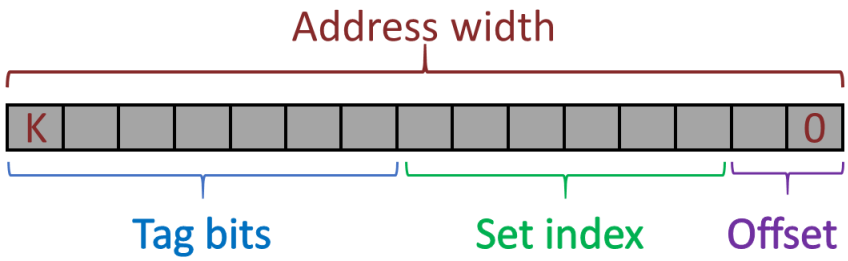

Figure A.2: Relation between address, tag, index and offset

Address bits are broken down to represent set index, offset and tag as shown in Figure A.2. Both data and instructions are byte-addressable. An important point to note is that there are certain limitations to the permitted configurations. Firstly, all the parameters are expected to be non-zero. Data width must be a power of two. The sum of offset bit-width index bit-width and tag size should be equal to the address width. Number of cores is always greater than or equal to one. Lastly, instruction bound must be lower than the maximum possible address.

A.1 Algorithm Description

A.1.1 Replacement Policy

The level-1 cache follows a pseudo-Least Recently Used(PLRU) algorithm when the cache set is full. Each set within the cache has a 3-bit LRU state variable. In the absence of a free block, the LRU state dictates which line is replaced. This relation is depicted in Table A.1, 'x' indicates 'don't care'. Additionally, every access to cache updates the state as shown in Table A.2, '-' indicates that the bit is unchanged.

Table A.1: Pseudo-LRU replacement policy

State	Replacement
00x	Line 0
01x	Line 1
1x0	Line 2
1x1	Line 3

Table A.2: Pseudo-LRU state update

Access	Next state
Line 0	11-
Line 1	10-
Line 2	0-1
Line 3	0-0

A.1.2 MESI Protocol

Each cache line has its own MESI state which is either Modified (M), Exclusive (E), Shared (S), or Invalid (I). The state diagram for the protocol is illustrated as Figure A.3.

Transitions to the left of states, represent processor side requests and the subsequent controller action on the shared bus. More precisely, cpu rd(read) and cpu wr(write) are requests from the core; bus rdx(read with intent to modify), bus rd(read-only), and invalidate are requests broadcast on the shared system bus. State transitions to the right, signify incoming bus requests and corresponding responses. Each transition in the state diagram has two components i.e. x/y. Here 'x' signifies the incoming request either from the core or another L-1 cache, and 'y' indicates the required action/message on the bus.

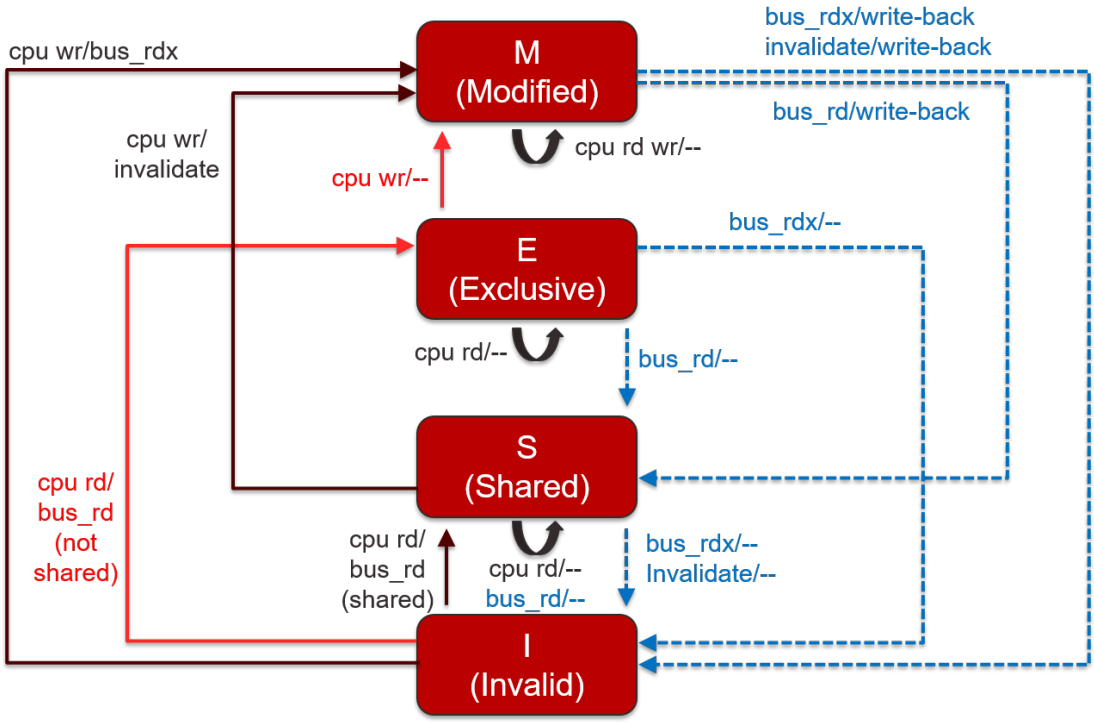

Figure A.3: MESI coherence protocol

A.2 Design Hierarchy

The multi-core L1 cache module houses a configurable number of uni-core cache blocks as shown in Figure A.4. It interacts with the CPU cores, arbiter and L2 cache.

It houses an multiplexer block which is necessary to avoid input-output signals. The multiplexer serves as the interconnect, routing signals based on the arbiter outputs. It selects the address, bus request and related signals based on processor grant. Likewise, the bus response is routed based on snoop grant signal from the arbiter. Each uni-core cache module has a separate instruction cache and data cache. A coherence protocol is unnecessary for instructions based on the assumption that cores do not write to an instruction address. Similarly, a write operation to I-cache is expected to be unacknowledged.

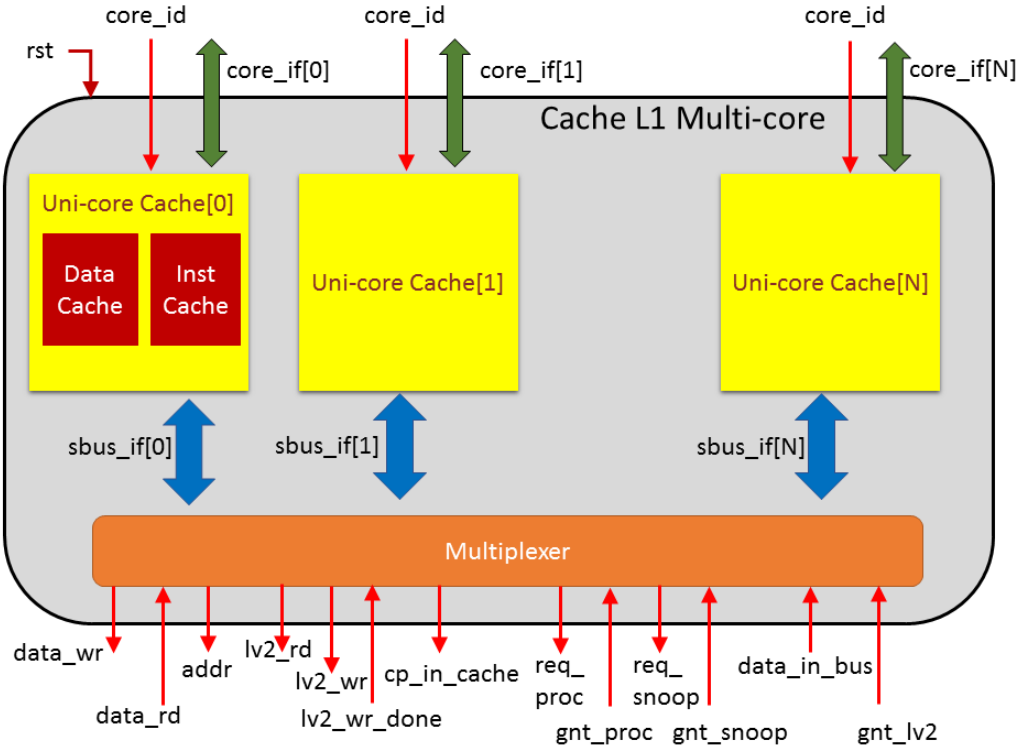

Figure A.4: Multi-core L1 cache design hierarchy

A.3 IO Interface

In this section, we indicate the IO interface for a configuration of 4-core cache with 32-bit address, 32-bit cache block, 16-bit tag, 14-bit set index and 2-bit offset. This configuration is the expected usage model. It is the main focus of the simulation exercise.

A.3.1 Uni-core Cache Interface

Table A.3: IO interface for uni-core cache

Signal	**Type**	**Description**
rst	Input	Active-high reset
core_id	Input(2)	Identifier for the cache
core_if		Interface for interaction with the core
clk	Input	Clock signal
cpu_rd	Input	CPU read request
data_in_bus_cpu_lv1	Output	Data valid signal for read request
cpu_wr	Input	CPU write request
cpu_wr_done	Output	Acknowledge signal for write operation
data_bus_cpu_lv1_wr	Input(32)	Data bus for write
data_bus_cpu_lv1_rd	Output(32)	Data bus for read
addr_bus_cpu_lv1	Input(32)	Address bus
sbus_if		Interface to system bus
in_data_bus_lv1_lv2	Input(32)	Data bus for incoming SBUS request
in_addr_bus_lv1_lv2	Input(32)	Address for incoming SBUS request
out_data_bus_lv1_lv2	Output(32)	Data for outgoing SBUS request
out_addr_bus_lv1_lv2	Output(32)	Address for outgoing SBUS request
lv2_rd	Output	Read to level-2 cache
lv2_wr	Output	Write to L2 cache
lv2_wr_done	Input	Write acknowledgment from L2
bus_rd_proc	Output	Outgoing bus read request
bus_rdx_proc	Output	Outgoing bus read request with intent to modify (write)
invalidate_proc	Output	Outgoing invalidate request
bus_rd_snoop	Input	Incoming bus read request
bus_rdx_snoop	Input	Incoming bus read request with intent to modify (write)

Table A.3: Continued

Signal	Type	Description
invalidate_snoop	Input	Incoming invalidate request
in_data_in_bus_lv1_lv2	Input	Response to outgoing read request
out_data_in_bus_lv1_lv2	Output	Response to snoop bus_rd or bus_rdx request
shared_local	Output	Signal to indicate if data block is shared
shared	Input	Signal to indicate if data supplied by snooping cache is shared
cp_in_cache	Output	Signal to L2 to ignore the read request as a copy exists in cache
invalidation_done	Output	Response to incoming invalidate request
all_invalidation_done	Input	Signal indicating if other caches have invalidated the data block
bus_lv1_lv2_req_proc	Output	Request for primary system bus access
bus_lv1_lv2_gnt_proc	Input	Grant of primary system bus access
bus_lv1_lv2_req_snoop	Output	Request for snoop system bus access
bus_lv1_lv2_gnt_snoop	Input	Grant of snoop system bus access
data_bus_lv1_lv2_wr	Output(32)	Data bus for write to L2
data_bus_lv1_lv2_rd	Input(32)	Data bus for read from L2
addr_bus_lv1_lv2	Output(32)	Address bus

A.3.2 Multi-core Cache Interface

A.4 Expected Behavior

In this section, we describe the sequence of signals for each type of CPU request. Note that the signal interaction is described at the uni-core cache level. The primary cache which has system bus access is called 'proc', and the snooping cache is labeled 'snoop'. Uni-core operation is defined for both the proc and snoop caches. Refer to Section A.6 to understand how uni-core signals are connected to each other.

A reset to the module is expected to completely clear the contents of all the uni-core caches. CPUs are expected to retain the request as logic high until serviced by the cache. Similarly, level-2 is required to retain the acknowledgment signal high until the multi-core

Table A.4: IO interface for multi-core cache

Signal	Type	Description
rst	Input	Active-high reset
core_if	Interface(4)	Interaction with the cores
bus_lv1_lv2_req_proc	Output(4)	Request for primary system bus access sent to arbiter
bus_lv1_lv2_gnt_proc	Input(4)	Grant of primary system bus access from arbiter
bus_lv1_lv2_req_snoop	Output(4)	Request for snoop system bus access sent to arbiter
bus_lv1_lv2_gnt_snoop	Input(4)	Grant of snoop system bus access from arbiter
lv2_rd	Output	Read to level-2 cache
lv2_wr	Output	Write to L2 cache
lv2_wr_done	Input	Write acknowledgment from L2
cp_in_cache	Output	Signal to L2 to ignore the read request as a copy exists in cache

L1 cache de-asserts the bus request.

A.4.1 CPU Read

A.4.1.1 Case 1: Read hit

If the block is hit in the L1 cache, then

- Data value is driven on data_bus_cpu_lv1_rd of that processor and data_in_bus_cpu_lv1 is asserted
- Signal bus_lv1_lv2_req_proc remains de-asserted

A.4.1.2 Case 2: Read miss

If the block is not present in the L1 cache, then we have two scenarios:

Scenario 1: Free block available in the set

- Bus access is requested (bus_lv1_lv2_req_proc is asserted high)

- Wait until access is granted (bus_lv1_lv2_gnt_proc is to be made high by arbiter)
- Once access granted, bus_rd_proc and lv2_rd is raised for data, whereas only lv_rd is asserted for instruction access. Concurrently, address of the requested block is put in out_addr_bus_lv1_lv2. This defines the outgoing request on the system bus.
- L2 cache or another L1 cache will provide the data on in_data_bus_lv1_lv2 and drive in_data_in_bus_lv1_lv2 high
- Appropriate cache line is updated with the data and corresponding MESI state

Following the above operation, the block will automatically hit in the cache. Therefore, the sequence described in Case 1 is carried out to complete the request.

Snooping cache:

Concurrently, on the snoop side, L1 caches with copies of the above request block (snoop hit) perform the following operations. Other caches with copies understand that a uni-core cache is requesting for read-only access to the block as bus_rd_snoop signal is asserted (which was made high by above mentioned proc side process). The multiplexer module connects the bus_rd_proc of the primary cache(proc) to the bus_rd_snoop inputs of the remaining uni-core cache modules.

- Signal cp_in_cache is asserted, asking L2 to ignore the current bus request
- Snoop bus access is requested (bus_lv1_lv2_req_snoop)
- If in_data_in_bus_lv1_lv2 is asserted and own snoop bus request is not granted, then bus_lv1_lv2_req_snoop is de-asserted immediately

Copy of block is in Shared/Exclusive state

- Signal shared_local is made high

- Data is put in out_data_bus_lv1_lv2 and out_data_in_bus_lv1_lv2 is made high
- MESI state is updated to Shared
- Signal bus_lv1_lv2_req_snoop is de-asserted

Copy of block is in Modified state

- Bus out_data_bus_lv1_lv2 is loaded with data from modified copy
- Signal lv2_wr is asserted to make level 2 cache update its value
- Wait for lv2_wr_done
- Signal shared_local is made high
- Signal out_data_in_bus_lv1_lv2 is made high
- MESI state is updated to Shared
- Signal bus_lv1_lv2_req_snoop is de-asserted

If no other level-1 cache has a copy, then level-2 cache provides the data.

Scenario 2: Free block not available in the set; replacement needed

PLRU algorithm determines the block to be evicted. If this block is Shared/Exclusive, then the MESI state is changed to Invalid as soon as proc grant is received. No additional message is relayed on the system bus. This is referred to as a silent eviction. However, if the block to be replaced is in Modified state, then the following sequence of actions are adopted.

- Signal bus_lv1_lv2_req_proc is asserted to logic high
- Address of the evicted block is generated from the tag and set index, and loaded into bus out_addr_bus_lv1_lv2

- Bus out_data_bus_lv1_lv2 is loaded with the dirty data
- Signal lv2_wr is made high requesting level 2 cache to update its value
- Once lv2_wr_done is made high by level 2 cache, block is assigned Invalid MESI state

These set of operations will free the L1 cache line of the evicted block, which in turn triggers the free block operation described above in Scenario 1.

A.4.2 CPU Write

Firstly, as soon as L1 cache receives a write request, bus_lv1_lv2_req_proc is made high.

A.4.2.1 Case 1: Write hit

When the block is hit, then the following operations are carried out depending on the block's MESI state:

Scenario 1: Block in Modified or Exclusive state

- Cache data is updated with the latest value
- MESI state is altered to Modified
- Signal cpu_wr_done is raised high
- Signal bus_lv1_lv2_req_proc is made low immediately

Scenario 2: Block in Shared state

- Wait for bus_lv1_lv2_gnt_proc to be asserted
- Address is loaded onto out_addr_bus_lv1_lv2

- Signal invalidate_proc is made high asking other level-1 caches to make their copy invalid
- When all such copies are invalidated, all_invalidation_done is made high
- Signal bus_lv1_lv2_req_proc is de-asserted
- Cache data is updated with the latest value
- MESI state is altered to Modified
- Signal cpu_wr_done is raised high

Snoop side for invalidation request

- If block isn't present, assert invalidation_done until all_invalidation_done is asserted
- If block is present, assert the shared_local signal; subsequently, invalidate the block in cache and assert invalidation_done until all_invalidation_done is high

A.4.2.2 Case 2: Write miss

The write miss case, similar to the read miss case, has two possibilities.

Scenario 1: Free block available

- Wait for bus_lv1_lv2_gnt_proc to be asserted
- Raise bus_rdx_proc and lv2_rd
- Drive the address on out_addr_bus_lv1_lv2
- Wait till level 2 cache provides the data by making data_in_bus_lv1_lv2 high. Note that data will always be provided by level 2 cache in this case (bus_rdx request)
- Once in_data_in_bus_lv1_lv2 is high, update the cache with the received data value and modify the MESI state to Exclusive

Following the above operation, the block will automatically hit in the cache. Therefore, the sequence described in Case 1 for Exclusive write hit is carried out to complete the write request.

Snooping cache:

On the snoop side, any cache with a copy of the relevant block must undertake the following actions. Firstly, cp_in_cache is asserted as soon as it is snoop hit for a incoming bus_rdx request. *Copy in Shared state*

- Signal shared_local is made high
- Cache copy is invalidated
- Signal shared_local is made low

Copy in Exclusive state

- Cache copy is invalidated

Copy in Modified state

- Signal bus_lv1_lv2_req_snoop is raised to ask for access to the bus
- Wait for bus_lv1_lv2_gnt_snoop to be high
- Bus out_data_bus_lv1_lv2 is loaded with data from modified copy
- Signal lv2_wr is asserted to make level 2 cache update its value
- Wait for lv2_wr_done to be high
- Invalidate the copy in cache
- De-assert bus_lv1_lv2_req_snoop

An important feature is that level-2 always provides data to a bus_rdx request. The snooping cache with a dirty copy merely writes the data back into level-2. Subsequently, level-2 provides the latest data value to the primary cache which made the bus_rdx request.

Scenario 2: Free block not available in the set; replacement needed

PLRU algorithm determines the block to be evicted. If this block is Shared/Exclusive, then the MESI state is changed to Invalid as soon as proc grant is received. No additional message is relayed on the system bus. This is referred to as a silent eviction. However, if the block to be replaced is in Modified state, then the following sequence of actions are adopted.

- Signal bus_lv1_lv2_req_proc is asserted to logic high
- Address of the evicted block is generated from the tag and set index, and loaded into bus out_addr_bus_lv1_lv2
- Bus out_data_bus_lv1_lv2 is loaded with the dirty data
- Signal lv2_wr is made high requesting level 2 cache to update its value
- Once lv2_wr_done is made high by level 2 cache, block is assigned Invalid MESI state

These set of operations will free the L1 cache line of the evicted block, which in turn triggers the free block operation described above in Scenario 1.

A.5 Timing Specification

We illustrate timing for typical behavior in this section. Signals mentioned are with reference to the primary cache i.e. proc cache, except when it is a snoop side scenario. In snoop side illustrations, signals are with reference to the snoop cache.

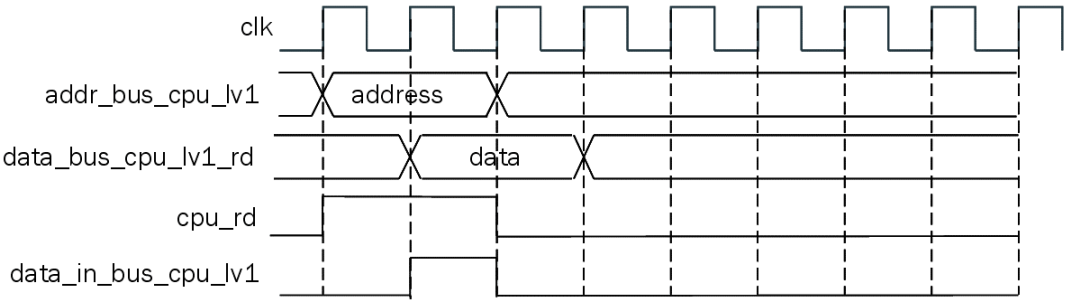

Figure A.5: Read hit scenario

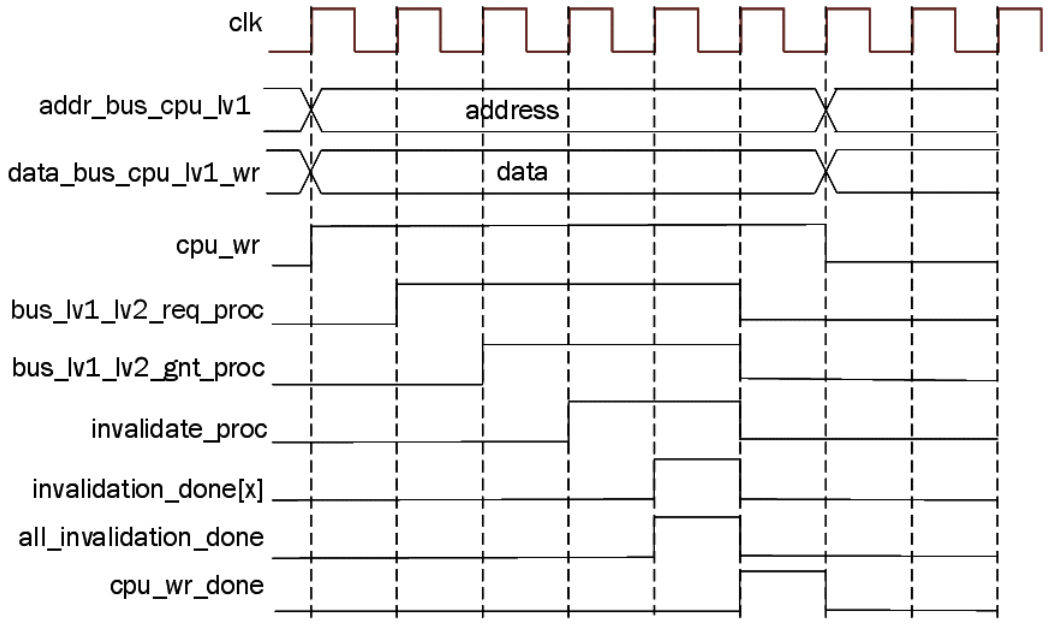

Figure A.6: Write hit scenario with shared block

A.6 Multiplexer Specification

The multiplexer module, within the multi-core cache, serves as the interconnect and glue-logic between uni-core L1 caches and the level-2 cache. The uni-core cache which

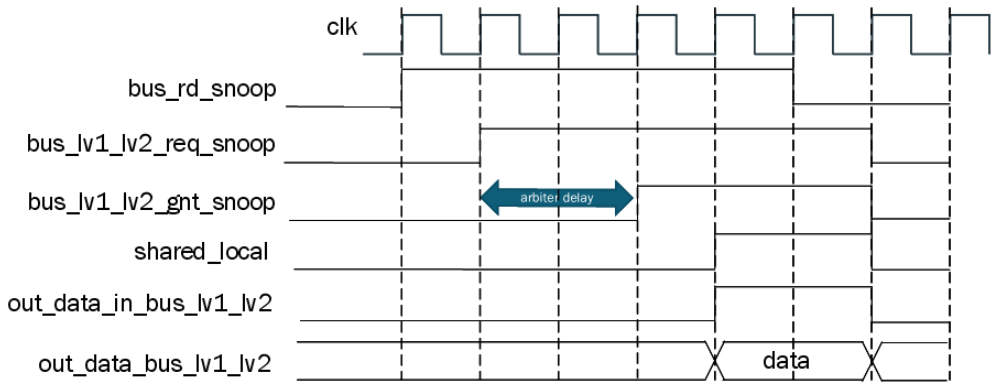

Figure A.7: Snoop scenario for bus rd with copy in shared/exclusive

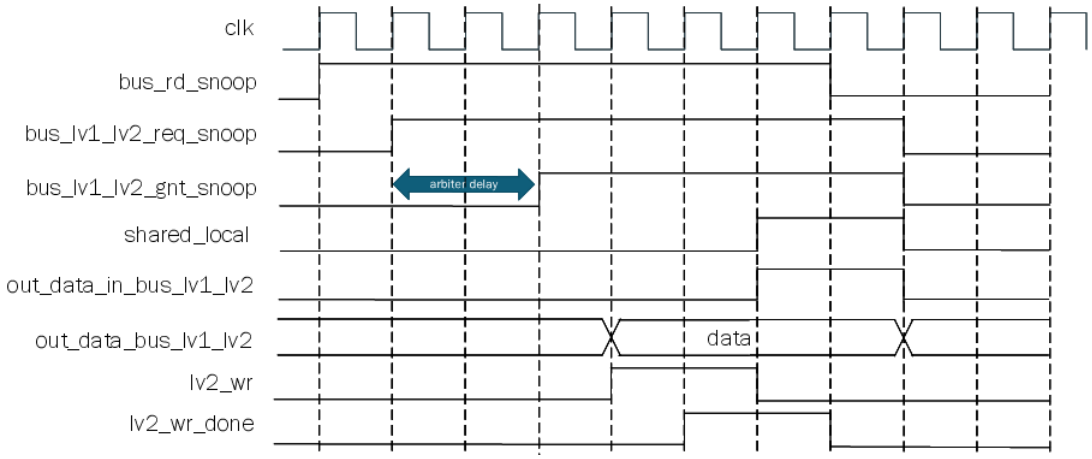

Figure A.8: Snoop scenario for bus rd with copy in modified

has obtained primary system bus access is referred to as 'proc', other L1 caches and L2 are known as the snooping caches. Here, we describe the routing of signals from the proc side to the snooping caches.

- Signal lv2_rd supplied to L2 is the logical OR of individual uni-core lv2_rd
- Signal lv2_wr supplied to L2 is the logical OR of individual uni-core lv2_wr

- Signal cp_in_cache supplied to L2 is the logical OR of individual uni-core cp_in_cache
- Signal shared is the logical OR of individual uni-core shared_local signals
- Signal all_invalidation_done is the logical AND of all snooping L1 cache's invalidation_done
- Output bus_rd_proc from the proc cache is routed to input bus_rd_snoop of all the snooping L1 caches
- Output bus_rdx_proc from the proc cache is routed to input bus_rdx_snoop of all the snooping L1 caches
- Output invalidate_proc from the proc cache is routed to input invalidate_snoop of all the snooping L1 caches
- Input address bus of snooping caches (L1 and L2) is always driven by the output out_addr_bus_lv1_lv2 of the proc cache
- Signal in_data_in_bus_lv1_lv2 is driven by the snooping cache with snoop grant access to the system bus
- Input data to all the caches (L1 and L2) is driven by output data bus of the snooping cache with snoop grant access to the shared bus. If none of the caches have snoop grant and lv2_wr is high, data is routed from proc's out_data_bus_lv1_lv2

A.7 Level-2 Cache

Last level cache is unified, and shared by all the cores. It does not distinguish between data and instruction address space. If cp_in_cache is logic high, LLC should ignore every request. An operation, can either be read(lv2_rd) or write(lv2_wr), and is processed only when cp_in_cache is low. MESI related signals bus_rd, bus_rdx, and invalidate do not

concern the level-2 cache. However, it plays a crucial role in the MESI protocol by supplying updated data value and processing write-backs. We define the timing specification in Figures A.9 and A.10.

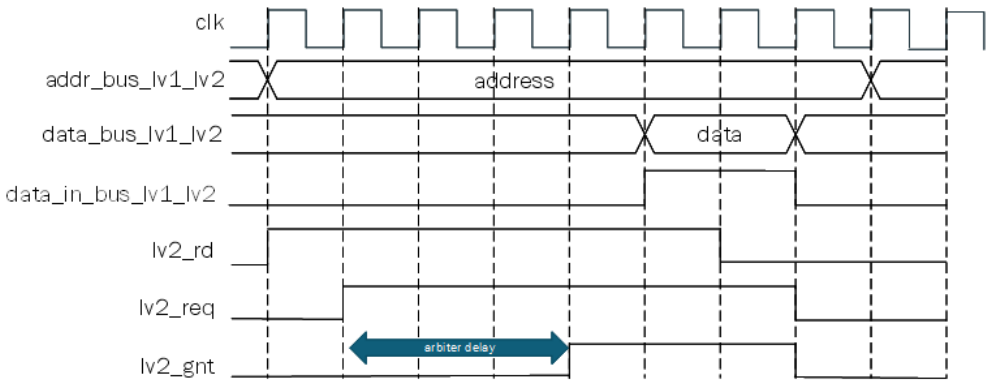

Figure A.9: Read serviced by level-2 cache

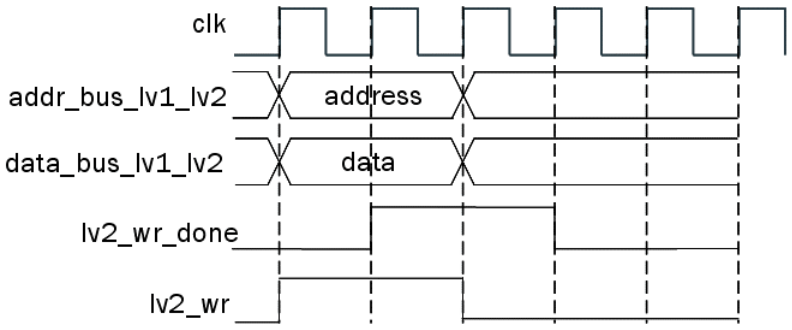

Figure A.10: Write back to level-2 cache

A.8 Arbiter Specification

This module arbitrates access to the shared system bus. There are two types of access to the common bus, namely processor side access and snoop side access. A simple round-robin scheme based on least-recently serviced cache is used for the processor side access. A fixed priority scheme is used for snoop access.

With respect to processor (primary) access, if only one request is observed, it is granted access on the next clock cycle. However, if more than one request is observed, the least-recently served cache is granted access first. L1-caches are expected to retain the request signal high, until serviced. A sample timing diagram is depicted in Figure A.11.

With respect to snoop access, request is granted based on the fixed priority. For the instance of a 4-core system, L1-cache0 > L1-cache1 > L1-cache2 > L1-cache3 > level-2 cache. LLC is always given the lowest priority. Snoop request is expected to be high, until one of the snooping cache is given access.

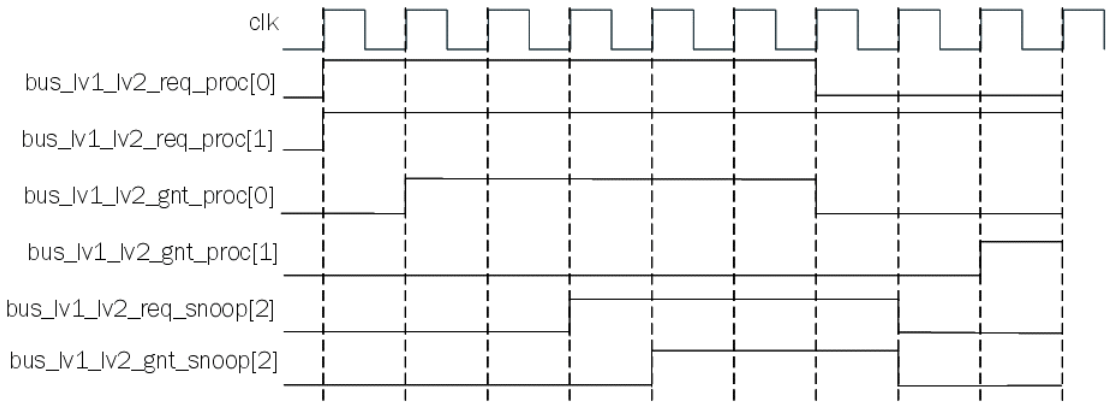

Figure A.11: Arbiter timing diagram

APPENDIX B

PROPERTIES

Properties form the crux of any formal property verification exercise. Here, we detail the cover statements, assumptions and assertions used in various stages of our verification effort.

B.1 Cover Points

Cover points are often mistakenly thought of as an afterthought to FPV. This can be disastrous especially in cases when assumptions over-constrain the design. Cover statements ensure that the set of typical behavior expected from the design is possible under the given set of constraints. They should ideally include: every specified operation, all legal input-output values, and each type of transaction. This section describes the cover points used to consistently sanity check our FPV environment. They are reviewed in every stage of FPV, whenever any of the assumptions are updated.

B.1.1 Uni-core Module

The following cover statements were defined for FPV of the uni-core cache module. Note that the state transition cover points were defined specifically for a particular cache line. Additionally, the state transition statements had two versions, namely an immediate version and an eventual variety. The immediate variant dictates that the state transition happened in a single clock cycle. An immediate transition from modified/exclusive to shared is improbable. Therefore, an immediate variant was not defined for these two transformations. The eventual version prescribed that the MESI state eventually changed from X to Y.

1. *CPU read transaction*

Simple read operation initiated by the CPU core, and completed with valid data returned by the associated L1 cache.

2. *CPU write transaction*

   Simple write operation initiated by the core and acknowledged by the associated L1-cache with write done signal.

3. *Outgoing bus_rd_proc transaction*

   A bus_rd request initiated by the uni-core module.

4. *Outgoing bus_rdx_proc transaction*

   A bus_rdx request initiated by the uni-core module.

5. *Outgoing invalidate_proc transaction*

   An invalidate request initiated by the module.

6. *Incoming bus_rd_snoop request*

   An incoming bus_rd request, serviced by the uni-core cache.

7. *Incoming bus_rdx_snoop request*

   An incoming bus_rdx request serviced by the module.

8. *Incoming invalidate_snoop request*

   An incoming invalidation request and the corresponding response from the uni-core cache.

9. *Instruction cache read miss scenario*

   An instruction read operation with access to the L2 cache over the system bus.

10. *Data cache read miss scenario*

    A data read operation serviced either by level-2 or another L1 cache.

11. *Invalid to modified state transition*

12. *Invalid to shared state transition*

13. *Invalid to exclusive state transition*

14. *Exclusive to invalid state transition*

15. *Exclusive to shared state transition*

16. *Exclusive to modified state transition*

17. *Shared to invalid state transition*

18. *Shared to exclusive state transition*

19. *Shared to modified state transition*

20. *Modified to invalid state transition*

21. *Modified to exclusive state transition*

22. *Modified to shared state transition*

Additionally, we analyzed the behavior for race conditions and contention between the CPU side and bus side requests with the following cover statements. Contention arises when CPU and bus-side operations simultaneously access the same data block. We classify the possible scenarios based on whether the CPU request arrived before or after the snoop-side message. If both requests are received at the exact same cycle, snoop-side request is assumed to have occurred first.

1. *CPU first, read hit followed by bus_rd*

   CPU request is received to a data block in the cache. Before the processor operation is completed, a bus_rd request for the same block is observed on the snoop-side bus.

2. *CPU first, read miss followed by bus_rd*

   CPU request to a data block not present in L1 cache. The uni-core cache has obtained the arbiter grant and received a copy of the data block into its cache. Before the CPU operation is completed, a bus_rd request for the same block is observed on the snoop side.

3. *CPU first, read hit followed by bus_rdx*

4. *CPU first, read miss followed by bus_rdx*

5. *CPU first, read hit followed by invalidate*

6. *CPU first, read miss followed by invalidate*

7. *CPU first, write hit followed by bus_rd*

8. *CPU first, write miss followed by bus_rd*

9. *CPU first, write hit followed by bus_rdx*

10. *CPU first, write miss followed by bus_rdx*

11. *CPU first, write hit followed by invalidate*

12. *CPU first, write miss followed by invalidate*

13. *Snoop first, bus_rd followed by CPU read*

    A snoop side bus_rd request is received. Before the snoop-side operation is serviced, a CPU read request is observed.

14. *Snoop first, bus_rd followed by CPU write*

15. *Snoop first, bus_rdx followed by CPU read*

16. *Snoop first, bus_rdx followed by CPU write*

17. *Snoop first, invalidate followed by CPU read*

18. *Snoop first, invalidate followed by CPU write*

B.1.2 Multi-core Module

FPV of the multi-core module re-used all the cover points identified in the uni-core effort. Additionally, the following cover statements were used to ensure typical behavior and accuracy in the arbiter and memory reference models.

1. Primary grant (bus_lv1_lv2_gnt_proc) is provided to each of the uni-core caches.

2. Snoop grant (bus_lv1_lv2_gnt_snoop) is provided to each of the uni-core caches.

3. All combinations of i and j, such that L1-cache$_i$ has proc grant and L1-cache$_j$ has snoop grant of the system bus.

4. Consecutive proc grants separated by a fixed delay

5. Consecutive snoop grants separated by a fixed delay

6. All primary bus (proc) requests asserted and eventually serviced by the arbiter.

7. Maximum possible snoop requests asserted and eventually transaction is completed.

B.2 Assumptions

Assumptions describe the legal behavior of the model inputs. They define the environment in which the DUV is expected to perform. They effectively constrain the problem space explored by the FPV tool. In order to facilitate full-proof verification, FPV was performed in increasing stages of complexity. Several simplifications, in the form of additional constraints, are introduced in early phases of our formal verification endeavor. We

clearly indicate these assumptions, which served to over constrain the design, as used only in specific stages of the verification process.

B.2.1 Uni-core Module

The generic assumptions common to all complexity stages are listed below:

1. Signals cpu_rd and cpu_wr cannot be high simultaneously in the same clock cycle.
2. Inputs on the CPU-lv1 interface are always legal i.e. no high impedance or unknown value.
3. Inputs on the system bus(sbus) interface are always legal.
4. Input address on the CPU-lv1 interface is stable during a CPU operation. In other words, address in unchanged whenever signal cpu_rd or cpu_wr is logic high.
5. Input data on the core interface is stable for the entire duration of a CPU write request.
6. Input address on the sbus interface is stable for the entire duration of an incoming snoop-side request.
7. Input data on the sbus interface is stable as long as data valid signal(in_data_in_bus_lv1_lv2) is asserted.
8. A read request to level-2 cache is serviced within a fixed number of clock cycles, say BUS_DATA_TIME.
9. A write request to level-2 cache is serviced within a fixed number of clock cycles, namely LV2_WR_RESP_TIME.
10. An outgoing invalidate request is serviced within a fixed delay, represented by the parameter INVALIDATE_TIME.

11. Signal cpu_rd is de-asserted on the cycle after data valid signal is received from the level-1 cache.

12. Signal cpu_wr is de-asserted on the clock cycle following a write acknowledgment from the uni-core cache.

13. An incoming snoop-side bus_rd request is de-asserted once data is provided by our uni-core module. In terms of signals, out_data_in_bus_lv1_lv2 triggers the de-assertion of bus_rd_snoop

14. An incoming snoop side bus_rdx request is de-asserted once data is provided by the level-2 cache.

15. An incoming invalidate snoop-side request is de-asserted on the cycle immediately after invalidation_done is driven high.

16. Signal shared is high as soon as shared_local is driven logic high.

17. Signal all_invalidation_done is asserted only if invalidation_done is high or an outgoing invalidate request is initiated by the uni-core module.

18. Processor grant(bus_lv1_lv2_gnt_proc) is provided only if the L1 cache has requested for system bus access(bus_lv1_lv2_req_proc).

19. Snoop grant(bus_lv1_lv2_gnt_snoop) is provided only if the L1 cache has requested for snooping access to system bus access(bus_lv1_lv2_req_snoop).

20. Signal in_data_in_bus_lv1_lv2 can be high only if lv2_rd is active.

21. Signal lv2_wr_done can be high only if lv2_wr is active.

22. Snoop side requests are mutually exclusive. Therefore, a maximum of one request from bus_rd_snoop, bus_rdx_snoop, and invalidate_snoop can be active at any point of time.

23. An ongoing snoop side request implies that processor grant (bus_lv1_lv2_gnt_proc) is not given to the uni-core module.

24. Signal in_data_in_bus_lv1_lv2 can be asserted only if the unicore cache has primary access of the system bus.

25. CPU read request remains high until serviced by L1 cache.

26. CPU write request remains asserted until acknowledged by L1 cache.

27. Snoop side bus_rd request must remain asserted until data is provided by the module if it has obtained snoop access of the system bus.

28. Snoop side bus_rd request should be high for a minimum of three clock cycles.

29. Snoop side bus_rdx request must remain high as long as cp_in_cache is driven high by the module under consideration.

30. Snoop side bus_rdx request should be high for a minimum of two clock cycles.

31. Snoop side invalidate request should remain high until the following cycle of a rising edge on all_invalidation_done.

32. Snoop side request implies that the address of a data block is present on the input address (in_addr_bus_lv1_lv2) of the shared bus.

33. An outgoing invalidate request is serviced within one clock cycle i.e. INVALIDATE_TIME is equal to 1 (This assumption is true as a result of arbiter specification).

34. An incoming snoop-side invalidate request depends on the current module's invalidation_done signal alone. In other words, all_invalidation_done signal is high whenever invalidation_done is high (This assumption is true as a result of arbiter specification).

B.2.1.1 Complexity stage 1

The following assumptions are specific to the first stage of the process. They serve to simplify the verification problem.

1. Allow write operations only to a data block
2. Processor grant (bus_lv1_lv2_gnt_proc) is provided immediately on the clock cycle following a primary access request (bus_lv1_lv2_req_proc) to the system bus.
3. Snoop grant (bus_lv1_lv2_gnt_snoop) is provided immediately on the clock cycle following a snoop access request (bus_lv1_lv2_req_snoop) to the system bus.
4. Data is provided on the system bus within a fixed latency (BUS_DATA_RESP = 2).

B.2.1.2 Complexity stage 2

1. Allow write operations to all blocks (instruction and data).
2. Processor grant (bus_lv1_lv2_gnt_proc) is provided immediately on the clock cycle following a primary access request (bus_lv1_lv2_req_proc) to the system bus.
3. Snoop grant (bus_lv1_lv2_gnt_snoop) is provided immediately on the clock cycle following a snoop access request (bus_lv1_lv2_req_snoop) to the system bus.
4. Data is provided on the system bus within a fixed latency (BUS_DATA_RESP = 3).

B.2.1.3 Complexity stage 3

1. Allow write operations to all blocks (instruction and data).

2. Processor grant (bus_lv1_lv2_gnt_proc) is provided within 45 clock cycle following a primary access request (bus_lv1_lv2_req_proc) to the system bus. Value of 45 is determined using worst case analysis for a 4-core system. If each of the other L1-caches perform a system bus operation including eviction and write-back, it would take each L1 cache exactly 15 cycles. This analysis is validated in multi-core FPV.

3. Snoop grant (bus_lv1_lv2_gnt_snoop) is provided immediately on the clock cycle following a snoop access request (bus_lv1_lv2_req_snoop) to the system bus only if the block is in modified/exclusive state.

4. Data is provided on the system bus within a fixed latency (BUS_DATA_RESP = 9). The value of 9 is determined using multi-core FPV.

B.2.2 Multi-core Module

Assumptions on inputs of the core_if in the uni-core setting are re-used in the multi-core environment. These assumptions, which define behavior of the CORE (CPU), are in accordance with the specification. All other assumptions used in the uni-core setting are converted into assertions in the instantiation environment. We use auxiliary SV code, compliant with the specification, to model behavior of level-2 cache and the round-robin arbiter. No additional constraints were necessary for the multi-core FPV effort.

Auxiliary SV code for level-2 and arbiter is provided below:

```
1  //------------------------------------------------------------------
2  // model for arbiter
3  //------------------------------------------------------------------
4  wire proc_gnt_any, snoop_gnt_any;
5
6  bit [`NO_OF_CORE-1 : 0 ] bus_lv1_lv2_gnt_proc_pre;
7  bit [`NO_OF_CORE-1 : 0 ] bus_lv1_lv2_gnt_snoop_pre;
8
9  bit[1:0] count[NO_OF_CORE-1:0];
```

```
10
11 assign proc_gnt_any  = | bus_lv1_lv2_gnt_proc  ;
12 assign snoop_gnt_any = | bus_lv1_lv2_gnt_snoop ;
13 int val, number, max;
14
15 function int gnt_number();
16         max = 0;
17         for (int i=0;i<`NO_OF_CORE;i++) begin
18                 if(bus_lv1_lv2_req_proc[i] == 1'b1) begin
19                         if(count[i] > max) begin
20                                 max = count[i];
21                                 number = i;
22                         end
23                 end
24         end
25         return number;
26 endfunction
27
28 always@(posedge clk or posedge rst) begin
29         if (rst) begin
30                 bus_lv1_lv2_gnt_proc_pre  <= `NO_OF_CORE'b0;
31                 for(int i=0; i<`NO_OF_CORE; i=i+1) begin
32                         count[i] <= i;
33                 end
34         end else if(~proc_gnt_any && (|bus_lv1_lv2_req_proc)) begin
35                 if(!(|(bus_lv1_lv2_req_proc & (bus_lv1_lv2_req_proc-1)))) begin
36                         bus_lv1_lv2_gnt_proc_pre <= bus_lv1_lv2_req_proc;
37                 end else begin
38                         bus_lv1_lv2_gnt_proc_pre <= `NO_OF_CORE'b0;
39                         val = gnt_number();
40                         bus_lv1_lv2_gnt_proc_pre[val] <= 1'b1;
41                         for (int i=0; i<`NO_OF_CORE; i=i+1)begin
42                                 if(i != val && count[i] < count[val]) begin
43                                         count[i] <= count[i]+1;
44                                 end
45                         end
46                         count[val] <= 2'b0;
47                 end
48         end else if(proc_gnt_any) begin
49                 bus_lv1_lv2_gnt_proc_pre <= bus_lv1_lv2_gnt_proc;
50         end else begin
51                 bus_lv1_lv2_gnt_proc_pre <= `NO_OF_CORE'b0;
52         end
53 end
54
55 // snoop grant
56 always@(posedge clk or posedge rst) begin
57         if(rst) begin
58                 bus_lv1_lv2_gnt_snoop_pre <= `NO_OF_CORE'b0;
59                 bus_lv1_lv2_gnt_lv2_pre <= 1'b0;
60         end else if(!proc_gnt_any) begin
61                 bus_lv1_lv2_gnt_snoop_pre <= `NO_OF_CORE'b0;
62                 bus_lv1_lv2_gnt_lv2_pre <= 1'b0;
63         end else begin
64                 bus_lv1_lv2_gnt_snoop_pre <= `NO_OF_CORE'b0;
65                 if(bus_lv1_lv2_req_snoop[0] == 1'b1)
66                         bus_lv1_lv2_gnt_snoop_pre[0] <= 1'b1;
67                 else if(bus_lv1_lv2_req_snoop[1] == 1'b1)
68                         bus_lv1_lv2_gnt_snoop_pre[1] <= 1'b1;
69                 else if(bus_lv1_lv2_req_snoop[2] == 1'b1)
70                         bus_lv1_lv2_gnt_snoop_pre[2] <= 1'b1;
71                 else if(bus_lv1_lv2_req_snoop[3] == 1'b1)
72                         bus_lv1_lv2_gnt_snoop_pre[3] <= 1'b1;
73                 else if(bus_lv1_lv2_req_lv2)
74                         bus_lv1_lv2_gnt_lv2_pre <= 1'b0;
75         end
76 end
77
78 mc_u_proc_gnt:  assume property(bus_lv1_lv2_gnt_proc == (bus_lv1_lv2_gnt_proc_pre &
       bus_lv1_lv2_req_proc));
```

```
79  mc_u_snoop_gnt: assume property(bus_lv1_lv2_gnt_snoop == (bus_lv1_lv2_gnt_snoop_pre &
        bus_lv1_lv2_req_snoop));
80  mc_u_lv2_gnt:   assume property(bus_lv1_lv2_gnt_lv2 == (bus_lv1_lv2_gnt_lv2_pre &
        bus_lv1_lv2_req_lv2));
81
82  //-------------------------------------------------------------
83  // model for memory
84  //-------------------------------------------------------------
85  bit [DATA_WID-1:0] memory[(1<<(ADDR_WID-OFFSET_WID))-1:0];
86  bit [DATA_WID-1:0] valid_data;
87  bit lv2_wr_done_pre, data_in_bus_lv1_lv2_pre, bus_lv1_lv2_req_lv2;
88
89  always@(posedge clk) begin
90          if(rst) begin
91                  for(int i=0; i<(`DL_ADDR_BOUND>>2); i=i+1) begin
92                          memory[i] <= `DATA_WID_LV1'hD;
93                  end
94                  valid_data <= `DATA_WID_LV1'h00;
95                  data_in_bus_lv1_lv2_pre <= 1'b0;
96                  lv2_wr_done_pre <= 1'b0;
97          end else if(lv2_rd && ~cp_in_cache && bus_lv1_lv2_gnt_lv2) begin
98                  valid_data <= memory[addr_bus_lv1_lv2[ADDR_WID-1:OFFSET_WID]];
99                  data_in_bus_lv1_lv2_pre <= 1'b1;
100         end else if(lv2_wr) begin
101                 memory[addr_bus_lv1_lv2[ADDR_WID-1:OFFSET_WID]] <=
                        data_bus_lv1_lv2_wr;
102                 lv2_wr_done_pre <= 1'b1;
103         end else begin
104                 data_in_bus_lv1_lv2_pre <= 1'b0;
105                 lv2_wr_done_pre <= 1'b0;
106         end
107 end
108
109 mc_u_lv2_data_rd: assume property(data_bus_lv1_lv2_rd == valid_data);
110 mc_u_data_in_bus: assume property(data_in_bus_lv1_lv2 == (data_in_bus_lv1_lv2_pre &
        lv2_rd));
111 mc_u_lv2_wr_done: assume property(lv2_wr_done == (lv2_wr_done_pre & lv2_wr));
112 mc_u_lv2_req1:    assume property((lv2_rd && !cp_in_cache) |=> bus_lv1_lv2_req_lv2);
113 mc_u_lv2_req2:    assume property((!lv2_rd || cp_in_cache)|=> !bus_lv1_lv2_req_lv2);
```

B.3 Assertions

Assertions are properties that we wish to prove about the design under legal stimulus. These statements are directly derived from the specification. Assertions are generally properties about the DUV's outputs, while assumptions describe the DUV's inputs. Assertions, which provide maximum insight, are end-to-end variants which only consider ports at the IO interface of the module. Alternatively, assertions can also be defined in terms of internal design signals. We use a mixture of end-to-end and internal assertions in our analysis.

B.3.1 Uni-core Module

Assertions defined for the uni-core FPV effort are described below.

B.3.1.1 CPU-lv1 interface

1. All outputs on the CPU-lv1 interface should be legal.
2. CPU read and write response signals are mutually exclusive.
3. Signal data_in_bus_cpu_lv1 is high only if CPU read operation is requested.
4. Write acknowledgment (cpu_wr_done) is given only if a CPU write operation is requested.
5. Output data bus is unchanged as long as data valid signal is logic high.

B.3.1.2 System bus interface

1. All outputs on the system bus interface should be legal.
2. Output address is unchanged when a outgoing request on the system bus, namely lv2_rd, lv2_wr, or invalidate_proc is asserted.
3. Data output to level-2 cache is stable during a level-2 write operation.
4. Data returned in response to a snoop-side bus_rd operation should be stable as long as out_data_in_bus_lv1_lv2 is logic high.
5. Signal cp_in_cache can go high only if there was a snoop side request (bus_rd or bus_rdx) on the previous clock cycle.
6. If there is an active snoop side request and lv2wr are asserted, signal cp_in_cache should be logic high.
7. If out_data_in_bus_lv1_lv2 is high, i.e. uni-core module is providing data to another L1 cache, shared_local and cp_in_cache signals must be asserted.

8. If an outgoing bus_rd or bus_rdx operation is on-going, lv2_rd should be asserted.

9. If a level-2 read operation is in progress, a rising edge on data valid signal should trigger a falling edge on level-2 read request in the next clock cycle.

10. If a level-2 write operation is in progress, a rising edge on lv2_wr_done signal should trigger a falling edge on level-2 write request in the next clock cycle.

11. If a bus read operation is in progress, a rising edge on data valid signal should trigger a falling edge on bus_rd_proc in the next clock cycle.

12. If a 'bus read with intent to modify' operation is in progress, a rising edge on data valid signal should trigger a falling edge on bus_rdx_proc in the next clock cycle.

13. If an outgoing invalidate operation is in progress, a rising edge on all_invalidation_done signal should trigger a falling edge on invalidate_proc in the next clock cycle.

14. Level-2 read and write signals cannot be asserted at the same instant.

15. The uni-core module can initiate a request (lv2_rd, bus_rd, bus_rdx, or invalidate) only if it has primary access granted to the system bus (bus_lv1_lv2_gnt_proc).

16. The uni-core module can initiate a level-2 write only if it has either primary or snoop access granted to the system bus.

17. DUV should request for primary access to the system bus only if any CPU operation is pending on the CPU-lv1 interface.

B.3.1.3 Liveness properties

1. A CPU read operation is completed within a fixed number of clock cycles, represented by CPU_RD_RESP_TIME.

2. A CPU write operation is completed within a fixed number of clock cycles, represented by CPU_WR_RESP_TIME.

B.3.1.4 MESI protocol

1. A state transition from invalid to modified implies that there was a CPU write to the relevant data block, which resulted in a outward bus_rdx request on the system bus.

2. A transition from invalid to shared implies that a CPU read operation resulted in a cache miss and shared signal was high in response to the outward bus_rd request.

3. A transition from invalid to exclusive state implies that a CPU read operation resulted in a cache miss and shared signal was low.

4. A shared to invalid transition implies either of two possibilities. First, the relevant cache block was evicted by a CPU operation. Second, an incoming snoop side request i.e. either bus_rdx_snoop or invalidate_snoop invalidated the data block.

5. A shared to exclusive transition should never be observed.

6. A change from shared to modified state implies that there was a CPU write operation, which resulted in an outward invalidation request.

7. A transition from exclusive to invalid state again implies two possibilities. First, the relevant cache block was evicted by a CPU operation. Second, an incoming snoop side request i.e. either bus_rdx_snoop or invalidate_snoop invalidated the data block.

8. An exclusive to shared transition implies an incoming bus_rd request was observed on the snoop-side system bus.

9. An exclusive to modified transition assures us that a CPU write operation resulted in a hit to this exclusive block.

10. A modified to invalid transition implies two possibilities. First, the relevant cache block was evicted by a CPU operation. Second, an incoming snoop side request i.e. either bus_rdx_snoop or invalidate_snoop invalidated the data block.

11. A modified to exclusive state transition is improbable.

12. A modified to shared state change is always triggered by a incoming bus_rd request on the common bus.

B.3.1.5 Coherence and memory consistency

The properties described in this section are crucial to guarantee coherence and memory consistency. They are inspired by the invariants defining coherence, presented in Section 1.1. A reference model is required to facilitate these assertions. This model maintains a copy of the valid data, which is referred as VCompData, within the verification component. Whenever data is supplied by the DUV, we essentially check for equivalence between the supplied data and VCompData. This copy is updated whenever there is a new CPU write operation to the uni-core module under consideration. Additionally, VCompData is also updated when an updated value is presented in response to a bus or level-2 read.

1. Data output by the uni-core module in response to a CPU read operation is equivalent to VCompData.

2. Data written back to level-2 cache by the DUV, in case of modified block eviction, is equal to VCompData.

3. Data written back to level-2 cache in response to an inward bus_rdx request is the same as VCompData.

4. Data provided by the DUV in response to a snoop-side bus_rd request is equivalent to VCompData.

B.3.1.6 Bug fixes

Several properties were added on-the-fly during the verification process. Specifically, the assertions in this section were added to reflect the proposed bug fixes. They are vital to ensure that design changes were implemented flawlessly.

A reference model was created to mimic the major architectural change propose d to overcome the contention issue between the processor and snoop-side requests. The reference model accurately recreates variables Conflict, CPUFirst, and SnoopFirst. Variable Conflict signified that a simultaneous CPU and snoop request to the same data block is observed. CPUFirst implies that the CPU operation must be given higher priority. Similarly SnoopFirst implies that the snoop-side request must be completed with higher priority. As a corollary, CPU operations are halted when SnoopFirst is asserted, and snoop-side requests are stalled when CPUFirst is logic high. This is clearly in line with the bug fix proposed in Section 5.2.2. The assertions relevant to this bug fix are listed below:

1. If there is a conflict and SnoopFirst has been asserted, the CPU operation should not receive a response.

2. If there is a conflict and CPUFirst has been asserted, the snoop-side request should not receive a response from this uni-core module.

Additionally, the following assertions were added to validate other independent bug fixes.

1. Signal data_in_bus_cpu_lv1 can be high only when input cpu_rd is high.

2. Signal cpu_wr_done can be high only when input cpu_wr is high.

3. Once the DUV has obtained primary access of the system bus, it should either request level-2 read, write or bus invalidate.

4. Once the DUV has obtained snoop side access of the system bus, it should either provide data or perform a write-back to level-2 cache.

B.3.2 Multi-core Module

All assertions from uni-core FPV were re-used in the multi-core module's property verification. All assumptions, except constraints on inputs of core_if, are redefined as assertions for multi-core FPV. This guarantees that the assumptions defined in uni-core verification are valid and safe. Additionally, we defined the following assertions specifically to guarantee accurate behavior of the arbiter model.

1. Outputs of the multi-core module are valid and legal

2. Address is stable during level-2 read and write

3. Data is unchanged during level-2 write

4. Primary (proc) grant to system bus is one-hot encoded. In other words, no more than one agent can hold primary access to the shared bus.

5. Snoop grant to the system bus is one-hot encoded. No more than a single agent is provided snoop grant.

6. A logic high on any snoop grant, implies that one bit on proc grant was asserted in the previous cycle.

7. A logic high on any proc request, implies that either proc grant will be asserted in the next cycle or one of the L1-caches will drop its request in the following cycle.

Most importantly, we defined the coherence and consistency assertion based on the data-value invariant. A CPU read on any of the cores should provide data on the previous successful CPU write. This is easily described as an end-to-end assertion in the multi-core model's FPV.