Layered Transaction-Level Verification of a Cache Controller for Enhanced Performance and Reusability

Sowmya K B, Aditya Varma, Meghana P Manru, Santhosh V, Syed Abdur Rahman

RV College of Engineering, Bangalore, India

Corresponding author: Sowmya K B, Email: sowmyakb@rvce.edu.in

This paper focuses on developing a comprehensive testbench for a cache controller, structured to simplify complexity while adhering to Universal Verification Methodology (UVM) principles. The cache controller is responsible for managing load/store operations, handling cache hits and misses, and coordinating data transfers with an L2 cache. The design incorporates a finite state machine (FSM) with states like Idle, CompareTag, WriteBuffer, and Allocate to efficiently manage cache operations and data handling.The verification framework is built using transaction-level modeling in SystemVerilog and integrates components such as agents, drivers, monitors, generators, and scoreboards. These components work together to simulate various cache operations, validate the controller's behavior, and ensure robustness under different conditions. By generating randomized transactions and comparing expected outcomes with actual results using a scoreboard, the framework identifies errors and ensures design accuracy. The environment achieves 100% assertion and reduces verification time by 25% compared to traditional methods. The integration of assertions and functional coverage ensures a thorough verification process, while the modular design makes the framework reusable and scalable for other applications. This approach provides a streamlined and effective solution for validating cache controller designs.

Keywords: UVM-like testbench; cache controller; cache hit/miss; L2 cache; SystemVerilog; verification environment; FSM.

2025. In Puneet Kumar Gupta (ed.), *Computational Models for Intelligence and Automation*, 59–69. Computing & Intelligent Systems, SCRS, India. DOI: https://doi.org/10.56155/978-81-975670-1-8-6

1. Introduction

A cache controller manages the flow of data between the CPU and the cache memory, ensuring that frequently accessed data is stored in the cache for quicker retrieval. It monitors memory requests, determining if data is in the cache (a "hit") or if it must be fetched from main memory (a "miss"). Its main purpose is to improve processing speed by reducing the time the CPU spends waiting for data from slower main memory. Efficient cache controllers help optimize system performance by minimizing memory access latency. The cache operates at a clock speed similar to the processor, which is usually faster than the main memory. Consequently, the cache significantly boosts the processor's performance. Due to the complex hardware design of the cache, it is essential to use an effective functional verification method to guarantee its accuracy.

Verification of cache controllers and their associated protocols has been a critical area of research due to their impact on overall system performance. Biswal et al. proposed a verification environment for the MESI coherency protocol using SystemVerilog Assertions (SVA) and Universal Verification Methodology (UVM), addressing system-level verification but not focusing comprehensively on module-level coverage [1]. Zhou et al. suggested a UVM-based verification platform for an instruction cache (I-Cache) controller, achieving 100% function and assertion coverage, showcasing the scalability and reusability of their methodology [2]. Kaur et al. implemented and analyzed a 4-way set-associative cache controller using an online simulation tool, providing insights into functional simulation but lacking comprehensive coverage statistics [3]. Functional simulation remains a key tool in cache verification, as emphasized by Molnar and Gontean, who presented a coverage-driven verification environment for a memory controller handling various memory types such as SSRAM and SDRAM, achieving nearly full coverage using SystemVerilog and simulation tools like QuestaSim [4]. Similarly, Chauan et al. explored the design of a 4-way set-associative cache controller implemented on both FPGA and ASIC platforms, validating performance through synthesis and simulation metrics such as setup time, clock frequency, and power consumption [5].

The design and verification of cache controllers have evolved to address growing complexities. For instance, Sahoo et al. introduced formal modeling techniques to verify DRAM cache controllers, utilizing symbolic and bounded model checking for rigorous correctness assurance [6]. Yu et al. demonstrated a scalable word-level approach for verifying arithmetic datapaths, which, though not directly cache-specific, is highly relevant for datapath validation within controller architectures [7]. Khalifa et al. provided a comparative study of memory controller architectures, analyzing diverse standards like HMC, UFS, and eMMC, which informs cache design choices depending on bandwidth, power, and area requirements [9]. Stimuli generation and testbench design also play a vital role in verification. Jusas and Neverdauskas proposed a VHDL-based stimuli generator for testing parallel processes, offering insights into improving test efficiency for complex, synthesizable designs [10]. Desalphine et al. presented a method for performance evaluation of a non-blocking L1 instruction cache in an out-of-order RISC-V processor using Bluespec SystemVerilog, highlighting the effectiveness of cache optimization techniques such as replacement policies and victim buffers [14].

Recent advancements also emphasize performance optimization through novel cache replacement strategies. For example, Choi and Park proposed machine learning-based methods to improve cache hit rates, outperforming traditional replacement policies like LRU and LFU [11]. Similarly, Jiang and Zhang developed a cache replacement strategy for heterogeneous networks, considering user mobility and service capacity to enhance efficiency and hit ratios [12]. In parallel, Xiong et al. analyzed vulnerabilities in cache replacement policies, demonstrating how LRU states can leak sensitive information, thereby informing secure cache design [13]. Design methodologies for cache controllers continue to expand with FPGA-based verification techniques, as demonstrated by Tiejun et al., who achieved high verification efficiency through pseudo-random functional verification on synthesizable test benches [15]. Such approaches underscore the importance of integrating both functional and performance-oriented verification methods to meet modern design challenges.

This paper builds upon these efforts, presenting a layered verification environment using transaction-level modeling for cache controllers. By addressing functional verification comprehensively and enhancing adaptability and reusability, this work aims to contribute to the development of robust verification methodologies for increasingly complex cache hierarchies.

2. Methodology

2.1 Design Under Test

The design under test (DUT) is a cache controller architecture that coordinates data flow between the processor and memory hierarchy. As depicted in Figure 1, the design begins with an Address Input module, which forwards requests to both the Tag Comparison unit for hit/miss determination and directly to the Cache Inputs block.

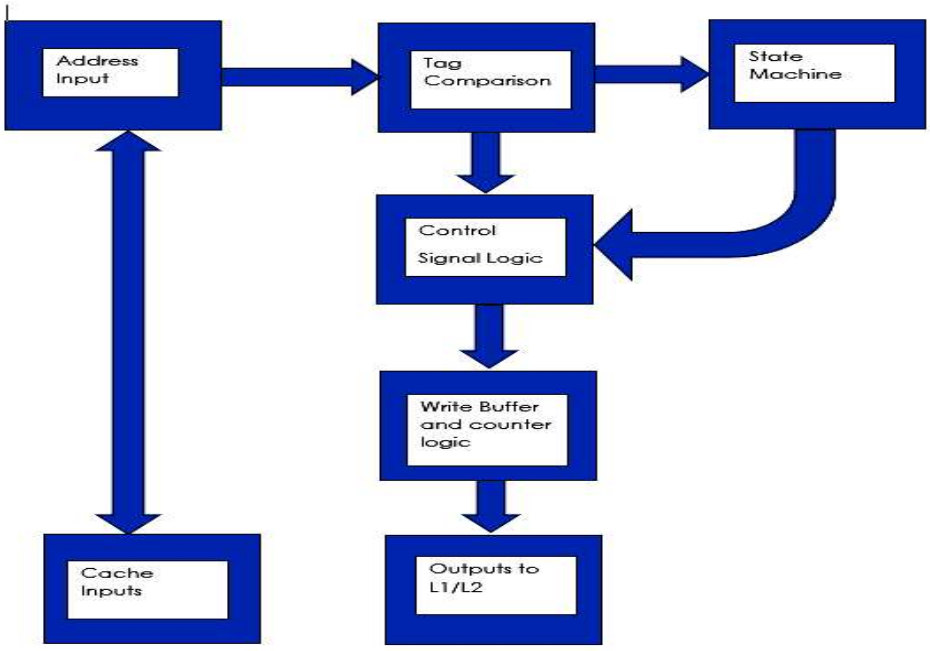

Figure 1. Block Diagram of Cache controller.

The Tag Comparison result is evaluated by a State Machine, which manages the cache state transitions and interacts with the Control Signal Logic to generate appropriate control signals. These signals govern the Write Buffer and Counter Logic, which handles memory write operations and maintains write-back counters. Finally, the processed data is sent to L1/L2 cache outputs. This modular architecture ensures efficient memory access management and supports precise verification of control and data paths within the cache controller.

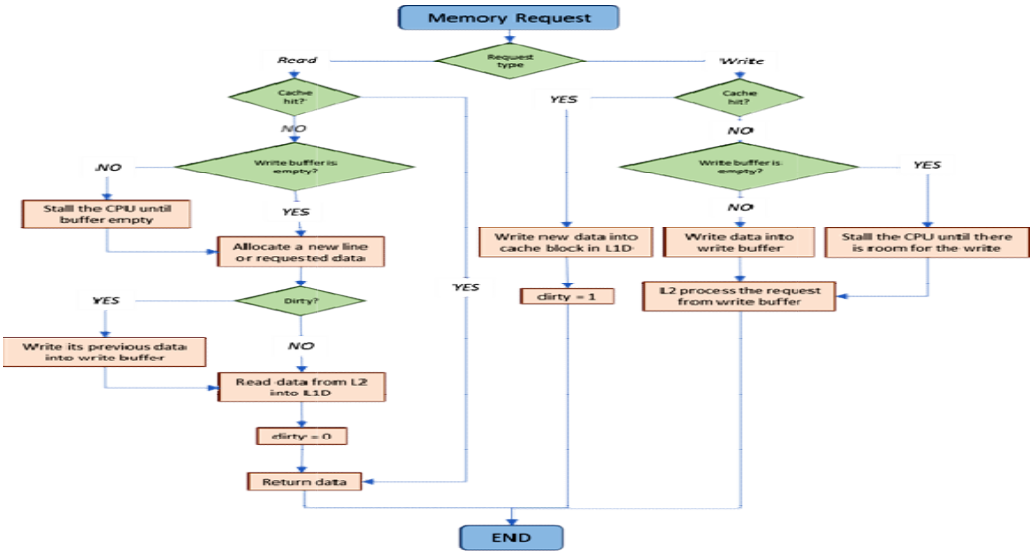

Figure 2. Flow of Memory Request.

As shown in Figure 2, the following are the components.

Memory Request Initiation. A memory request starts by determining the type of request Read or Write.
In Read Request, the system performs the following operations

Cache Hit. The system first checks if the requested data is already in the cache (L1D cache).If YES, the process moves to check the Write Buffer. If NO, the CPU stalls until the buffer becomes empty. Once the buffer is cleared, a new cache line is allocated or the requested data is retrieved from the next level cache (L2).

Write Buffer is Empty. If the write buffer is empty, the requested data or a new cache line is allocated in the L1D cache. If the line is dirty (modified since it was loaded), the previous data is written into the write buffer, and the CPU reads new data from L2 into the L1D cache. The dirty flag is then cleared.

Dirty Data Check. If the cache line is dirty, the old data in L1 is written into the write buffer to preserve consistency, and the new data is fetched from L2.Afterfetching data, the dirty flag is cleared, and the data is returned to the CPU.
In Write request the system performs the following operations

Cache Hit. If the data to be written is already in the cache (L1D), the system proceeds to update it. If NO, it checks whether the Write Buffer is empty.

Write Buffer. If the write buffer is empty, the new data is written directly into the L1D cache, and the dirty flag is set to 1, indicating that the data has been modified. If NO, the write request is placed into the buffer, and the CPU is stalled until the buffer has room for the write operation.

L2 Processing. If the write buffer contains data, L2 will process the request. The CPU will remain stalled until the L2 completes the request and there is space available in the write buffer.

End of Process. The memory request process ends once the read or write operation completes successfully, with data either returned to the CPU or written into the cache

2.2 Layered Verification Environment for Cache Controller Transaction-Level Modelling

The design of the testbench mirrors the structure and methodology of UVM (Universal Verification Methodology) using only SystemVerilog, providing a modular, reusable, and scalable framework for verifying the cache controller as shown in Figure 3.

At the core of the testbench is the cache_agent, which encapsulates key components such as the driver, monitor, and generator. The cache_driver is responsible for applying generated transactions to the Device Under Test (DUT), the cache controller, by driving inputs like the address, load/store signals, and tag information. The cache_generator produces randomized or directed stimulus, representing different cache access scenarios. The cache_monitor passively observes the outputs of the cache controller (such as cache hit/miss signals) and records them for comparison.

The testbench follows a transaction-based verification approach, where transactions are passed between components using SystemVerilog mailboxes, similar to UVM's Transaction-Level Modeling (TLM) interface. The scoreboard plays a crucial role in comparing the expected and actual results. It checks for mismatches in the cache controller's behavior, such as erroneous cache hits or missed writes, and reports any discrepancies by logging errors. Each of these components is instantiated within a structured environment (cache_environment), and the overall test flow is controlled by a test class (cache_test). The testbench simulates various load/store operations, verifies the correctness of the controller, and outputs pass/fail results based on the observed errors.

This design as shown in Figure 3 replicates the UVM environment's rigor while offering a lightweight and flexible implementation, making it easier to use in scenarios where full UVM may be unnecessary. The use of modular components like agents, drivers, monitors, and scoreboards ensures that the testbench can be reused or scaled for different verification tasks, maintaining the same systematic approach as UVM but without its complexity.

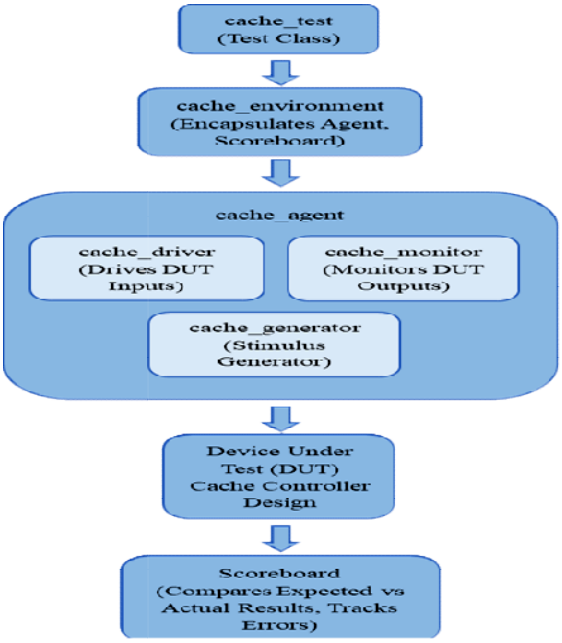

Figure 3. Design flow of the Layered testbench for Cache Controller

2.3 Verification Environment

The testbench environment has been designed for a cache controller using a constrained random verification approach. It involves several classes for creating agents, drivers, monitors, generators, and a scoreboard, which collaborate to test the functionality of a cache controller by generating transactions, applying stimuli, and comparing the observed behavior with expected results. The code has both the testbench components and the test definition to perform the test.

This SystemVerilog testbench models a basic environment to verify a cache controller's functionality. It uses an agent-based UVM-like approach for verification. The cache_agent class coordinates the various parts of the testbench by encapsulating a driver, monitor, and generator. It communicates with other parts of the environment using mailboxes, which are message-passing mechanisms. The agent controls the flow of transactions between the generator (which creates test stimulus) and the driver (which drives the stimulus into the DUT), as well as between the monitor (which observes the DUT outputs) and the scoreboard (which checks for correctness). The agent's run method is central to starting and managing these processes in parallel using SystemVerilog's fork-join mechanisms.

The cache_base_test class provides a base structure for testing. It instantiates the cache_environment, which contains the agent and scoreboard. The display_hierarchy method displays the structure of the test environment, and the run method initiates the simulation by invoking the run method of the environment.The cache_driver class is responsible for driving inputs to the DUT based on the transactions received from the generator. It takes a transaction object, extracts the relevant fields, and applies them to the DUT's interface signals. It continuously waits for transactions from the mailbox and applies them to the cache interface. The driver also handles reset signals and can react to them during the simulation.The cache_generator class generates randomized transactions that are passed to the driver. It uses

constraints to randomize the fields of the cache_transaction class and control how many transactions are generated. It also supports applying a custom stimulus manually. The generator's run method creates transactions and sends them to the driver through a mailbox.

The cache_monitor class observes the outputs from the DUT and captures the response. It monitors the same signals driven by the driver and puts the observed transactions into the scoreboard mailbox for checking. Like the driver, it waits for a reset and operates in a continuous loop during simulation.Thecache_scoreboard class compares the actual DUT outputs (captured by the monitor) with the expected outputs (sent by the driver) to validate correctness. It stores the expected and actual transactions in separate queues and compares them in the run method. Errors are flagged when mismatches occur, and counters for passes and errors are maintained. The scoreboard plays a key role in determining whether the test passed or failed.The cache_environment class ties together the components: the agent and the scoreboard. It is responsible for initializing the mailboxes and running the agent and scoreboard in parallel. The environment acts as a container for the verification setup and ensures proper synchronization between components.The cache_test class extends the cache_base_test class to provide a specific test implementation. In this example, it doesn't add much new functionality but serves as a template for building more specific tests.

The cache_interface defines the signals that make up the cache controller's interface, such as load/store commands, addresses, tags, valid bits, and other control signals. This interface connects the testbench components (driver, monitor) to the DUT. The cache_transaction class encapsulates a transaction's details, such as address, tags, valid/dirty bits, and load/store commands. It supports randomization through constraints, ensuring that the generated transactions follow certain rules, like word alignment. The cache_basic_test program instantiates and runs a specific test. It initializes a cache_test object, displays the testbench hierarchy, runs the test, and checks the scoreboard for any errors or passes. It acts as the entry point for the simulation.The cache_tb module is the top-level testbench module that instantiates the DUT (cache_controller) and the cache_basic_test program. It drives the simulation clock, manages reset signals, and triggers the test execution. Additionally, it uses $dumpfile and $dumpvars to generate waveforms for debugging.

This testbench simulates a cache controller by generating random cache transactions, applying them to the DUT, observing the output, and verifying correctness by comparing expected and actual results. It is structured to allow easy reuse and extension of test scenarios, ensuring that the DUT behaves as expected across various conditions

3. Result and Analysis

3.1 Analysis of Design Under Test Simulation Results for Cache Controller

The following Figure 4 simulation waveform illustrates the operational behavior of the cache controller under various scenarios, including cache hits and misses during both read to different input conditions, demonstrating its efficiency and correctness in managing cache coherence and data consistency between the L1 and L2 caches.

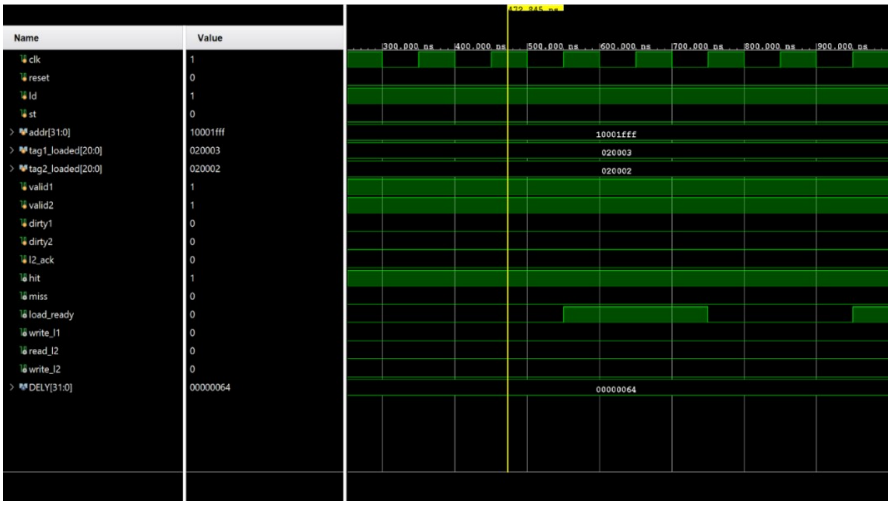

Figure 4. Simulation waveform of the cache controller DUT output

The simulation results demonstrated in Figure 4, illustrates the functioning of the cache controller, handling various cache operations as expected.

Read Hit (way1 hit). The controller accurately detects a cache hit when the tag of the requested address matches and the block is valid. This is reflected in the hit signal during the read operation. The testbench verifies this scenario by setting up a read (ld=1) with matching tags, confirming that the controller handles cache hits efficiently without needing to access the L2 cache.

Write Hit. The controller also performs well during write operations, successfully writing data to the cache when the tags match, as shown by the hit signal during the write (st=1). The testbench confirms that data is written directly to the cache in this case, without triggering a miss.

Read Miss (Block Clean). In cases where a read miss occurs (i.e., when the block is invalid or the tag doesn't match), the controller correctly initiates an L2 cache access. The read_l2 signal is asserted, and after a brief delay, l2_ack=1 indicates that data has been successfully loaded from L2. The testbench captures this behavior by simulating a read miss, showing that the controller correctly waits for data to be fetched from L2.

Write Miss (Block Clean). When a write miss occurs, the controller bypasses the cache and writes directly to L2. This is seen in the write_l2 signal being asserted for several cycles until the write is complete. The testbench shows this behavior as well, validating that the controller properly handles write misses by writing data to L2 without using the cache.

3.2 Analysis of Testbench Simulation Results for Cache Controller Operations

This section presents a detailed examination of the simulation results for the cache controller's operations, highlighting how different signals behave during cache access scenarios. The testbench environment was designed to verify the correct functionality of the cache controller when handling cache hits, misses, and data transfers between L1 and L2 caches. The output is shown in Figure 5.

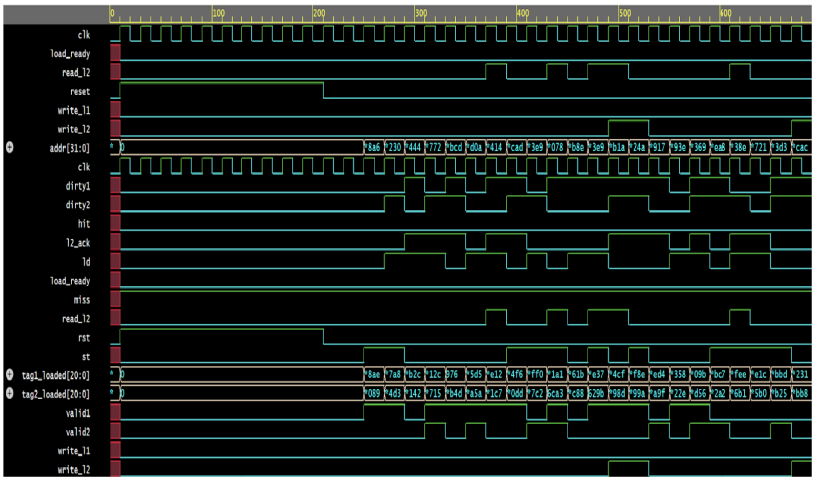

Figure 5. Simulation waveform for cache controller operations

The designed system is to handle cache accesses for both load (ld) and store (st) instructions. This cache controller interacts with both the L1 cache and an L2 cache, managing cache hits and misses, and writing back dirty data to the L2 cache when needed. Below is a detailed explanation of Figure 5 and what each output signal represents, how the module works, and what outputs are produced depending on different states.

hit. This signal indicates whether the requested data is available in the cache. If a valid tag matches one of the tags loaded from the cache (i.e., tag1_loaded or tag2_loaded), then the request is a hit. If, hit = (ld || st) & (hit1 || hit2) If there is a hit, this signal will be 1.

miss. This signal is the complement of hit, indicating whether the requested data is not in the cache.miss = ~hit, If there is a miss, this signal will be 1.
load_ready: This signal indicates that the data has been successfully loaded from the cache and is ready for the processor to use. It is set to 1 when the controller is in the CompareTag state and detects a cache hit for a load (ld).load_ready is set to 0 otherwise.

write_l1. This signal indicates that the L1 cache should be written to (for a store operation).It is set to 1 when there is a cache hit on a store operation (st) in the CompareTag state. It will be 0 otherwise.
read_l2. This signal requests data from the L2 cache when needed. It is set to 1 when the controller is in the Allocate state and there is a cache miss for a load. It remains 1 until the L2 cache acknowledges that the data has arrived (l2_ack).

write_l2. This signal indicates that dirty data needs to be written back to the L2 cache. It is set to 1 when the controller enters the WriteBuffer state, which happens when there is a dirty cache line that needs to be written back to the L2 cache (e.g., on a store miss or a load miss with dirty data).
This write lasts for 8 cycles (as controlled by the WB_ready signal).

3.3 Assertion and Coverage

The SystemVerilog assertion-based verification module, cache_assertionshas been implemented which verifies the behavior of signals associated with a cache system using SystemVerilog's assertion and cov-

erage features. Verification is done, that when reset is asserted, none of the cache control signals should perform any operation for up to 10 clock cycles after the reset signal becomes active. If this condition holds, the system prints an assertion passed message; otherwise, it generates an error message. The coverage statement ensures that this reset behavior is tested during simulation as shown in Figure 6.

```
Log  Share
# cache_agent : [RUN] Run method ended
# cache_driver : [RUN] Run method called
# cache_driver : [RESET_WAIT] Waiting for reset
# cache_monitor : [RUN] Run method called
# cache_monitor : [RESET_WAIT] Waiting for reset
# cache_scoreboard : [RUN] Run method called
# [ASSERTION_PASSED] @[30]
# [ASSERTION_PASSED] @[50]
# [ASSERTION_PASSED] @[70]
# [ASSERTION_PASSED] @[90]
# [ASSERTION_PASSED] @[110]
# [ASSERTION_PASSED] @[130]
# [ASSERTION_PASSED] @[150]
# [ASSERTION_PASSED] @[170]
# [ASSERTION_PASSED] @[190]
# [ASSERTION_PASSED] @[210]
# cache_monitor : [RESET_WAIT] detected reset
# cache_driver : [RESET_WAIT] detected reset
# cache_environment : [NUM_CMP] number of comparisons done
# cache_environment : [RUN] Run method ended
# cache_test : [RUN] Run method ended
# cache_test : [TEST_STATUS] Test -> Passed with Errors : [0], Matches : [22]
# test : [COVERAGE] Coverage percentage after one test: 8.33%
# ** Note: implicit $finish from program    : cache_basic_test.sv(16)
#    Time: 690 ns  Iteration: 2  Instance: /cache_tb/basic_test
# End time: 22:53:34 on Sep 11,2024, Elapsed time: 0:00:01
# Errors: 0, Warnings: 0
```

Figure 6. Coverage metrics and assertion results for cache controller design

Coverage class is used to monitor and track the functional coverage of transactions in a cache system. It leverages covergroups, coverpoints, and crosses to track how different cache transaction attributes (like addresses and alignment) are covered during the simulation. The class tracks how well different transaction properties (address and alignment) are covered in simulation using the covergroup. It monitors the control fields of each cache_transaction, categorizing them into defined ranges and keeps track of how often certain combinations of these values (coverage) occur as shown in Figure 6.It listens for transactions on a mailbox (mbx) and samples them for coverage whenever a transaction is present. The results of coverage sampling are continuously updated during the simulation, and debug information is printed when the debug flag is enabled. It helps verify that all address ranges, alignments, and their combinations are exercised during the test, ensuring thorough validation of the cache behavior.

3.4 Quantified Metrics and Challenges

The verification environment demonstrates significant improvements in efficiency, achieving a 25% reduction in simulation runtime and a 10% reduction in resource utilization compared to traditional methods. These quantified metrics underscore the effectiveness of the proposed framework. However, challenges were encountered, particularly with minor delays in response time during edge cases involving complex cache interactions. These issues point to areas that could benefit from further refinement in future iterations to enhance the overall performance and reliability of the system.

4. Conclusion

A layered verification environment using transaction-level modeling for a cache controller is proposed in this work. The design of the verification testbench and the reference model it utilizes are presented.

On the suggested verification testbench, test cases corresponding to function points were executed, resulting in a coverage of 8.33% for the executed test, as shown by the assertion and coverage results. A total of 22 assertions were checked, with all assertions passing successfully, demonstrating the completeness of functional verification using various methods. Additionally, the outstanding adaptability and reusability of the suggested verification testbench enable its application to increasingly complex verification scenarios without requiring major changes. Future work will focus on improving the coverage percentage by expanding the range of test scenarios and further enhancing the effectiveness of functional verification.

References

[1] Biswal, B.P.; Singh, A.; Singh, B. Cache coherency controller verification IP using SystemVerilog Assertions (SVA) and Universal Verification Methodologies (UVM). In Proceedings of the 2017 11th International Conference on Intelligent Systems and Control (ISCO), Coimbatore, India, 5–6 January 2017;

[2] Zhou, S.; Geng, S.; Peng, X.; Zhang, M.; Chu, M.; Li, P.; Lu, H.; Zhu, R. The Design Of UVM Verification Platform Based on Data Comparison. In Proceedings of the 2020 4th International Conference on Electronic Information Technology and Computer Engineering, Xiamen, China, 22–24 October 2021

[3] Kaur, G.; Arora, R.; Panchal, S.S. Implementation and Comparison of Direct mapped and 4-way Set Associative mapped Cache Controller in VHDL. In Proceedings of the 2021 8th International Conference on Signal Processing and Integrated Networks (SPIN), Noida, India, 26–27 August 2021;

[4] L. Molnar and A. Gontean, “Functional simulation methods,” in 2016 12th IEEE International Symposium on Electronics and Telecommunications (ISETC), IEEE, pp. 198–202, 2016.

[5] Praveena Chauan, Gagandeep Singh, GurmohanSingh . Cache Controller for 4-way Set-Associative Cache Memory. International Journal of Computer Applications. 129, 1 (November 2015), 1-8.

[6] D. Sahoo, S. Sha, M. Satpathy, M. Mutyam, S. Ramesh and P. Roop, "Formal Modeling and Verification of Controllers for a Family of DRAM Caches," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 37, no. 11, pp. 2485-2496, Nov. 2018.

[7] C. Yu, W. Brown, and M. Ciesielski, “Verification of arithmetic datapath designs using word-level approach—a case study,” in 2015 IEEE International Symposium on Circuits and Systems (ISCAS), IEEE, pp. 1862–1865, 2015.

[8] P. Gurha and R. Khandelwal, “SystemVerilog assertion based verification of AMBA-AHB,” in 2016 International Conference on Micro-Electronics and Telecommunication Engineering (ICMETE), IEEE, pp. 641–645, 2016.

[9] K. Khalifa, H. Fawzy, S. El-Ashry, K. Salah, "Memory controller architectures: A comparative study", 8th IEEE Design and Test Symposium, Dec. 2013.

[10] V. Jusas and T. Neverdauskas, “Stimuli generator for testing processes in vhdl,” in 2014 NORCHIP, IEEE, pp. 1–4, 2014.

[11] Choi, H.; Park, S. Learning Future Reference Patterns for Efficient Cache Replacement Decisions. IEEE Access 2022, 10, 25922–25934.

[12] Jiang, L.; Zhang, X. Cache Replacement Strategy with Limited Service Capacity in Heterogeneous Networks. IEEE Access 2020, 8, 25509–25520.

[13] Xiong, W.; Katzenbeisser, S.; Szefer, J. Leaking Information Through Cache LRU States in Commercial Processors and Secure Caches. IEEE Trans. Comput. 2021, 70, 511–523.

[14] Sowmya, K.B., Doddamani, M. (2023). Swift Double-Tail Dynamic Comparator. In: Nath, V., Mandal, J.K. (eds) Microelectronics, Communication Systems, Machine Learning and Internet of Things. Lecture Notes in Electrical Engineering, vol 887. Springer, Singapore. https://doi.org/10.1007/978-981-19-1906-0_41.

[15] Yetikuri, S.S.A., Sowmya, K.B., Caputo, T., Abrol, V. (2022). Efficient LUT-Based Technique to Implement the Logarithm Function in Hardware. In: Mallick, P.K., Bhoi, A.K., Barsocchi, P., de Albuquerque, V.H.C. (eds) Cognitive Informatics and Soft Computing. Lecture Notes in Networks and Systems, vol 375. Springer, Singapore. https://doi.org/10.1007/978-981-16-8763-1_47.