Indonesian Journal of Electrical Engineering and Informatics (IJEEI)
Vol. 11, No. 4, December 2023, pp. 1051~1063
ISSN: 2089-3272, DOI: 10.52549/ijeei.v11i4.5014

Design and Performance Analysis of a Fast 4-Way Set Associative Cache Controller using Tree Pseudo Least Recently Used Algorithm

Mohamed Alfian Al-Zikry Hazlan[1], Teddy Surya Gunawan[2], Mashkuri Yaacob[3], Mira Kartiwi[4], Fatchul Arifin[5]

[1,2,3]Department of Electrical and Computer Engineering, International Islamic University Malaysia, Kuala Lumpur, Malaysia
[4]Department of Information Systems, International Islamic University Malaysia, Kuala Lumpur, Malaysia
[5]Department of Electronic and Informatics Engineering, Universitas Negeri Yogyakarta, Yogyakarta, Indonesia

Article Info

Article history:

Received Aug 11, 2023
Revised Dec 9, 2023
Accepted Dec 20, 2023

Keywords:

Cache controller
VHDL-designed
Finite State Machine
4-way set associative cache
Tree PLRU algorithm

ABSTRACT

In the realm of modern computing, cache memory serves as an essential intermediary, mitigating the speed disparity between rapid processors and slower main memory. Central to this study is the development of an innovative cache controller for a 4-way set associative cache, meticulously crafted using VHDL and structured as a Finite State Machine. This controller efficiently oversees a cache of 256 bytes, with each block encompassing 128 bits or 16 bytes, organized into four sets containing four lines each. A key feature of this design is the incorporation of the Tree Pseudo Least Recently Used (PLRU) algorithm for cache replacement, a strategic choice aimed at optimizing cache performance. The effectiveness of this controller was rigorously evaluated using ModelSim, which generated a comprehensive timing diagram to validate the design's functionality, especially when integrated with a segmented main memory of four 1KB banks. The results from this evaluation were promising, showcasing precise logic outputs within the timing diagram. Operational efficiency was evidenced by the controller's swift processing speeds: read hits were completed in a mere three cycles, read misses in five and a half cycles, and both write hits and misses in three and a half cycles. These findings highlight the controller's capability to enhance cache memory efficiency, striking a balance between the complexities of set-associative mapping and the need for optimized performance in contemporary computing systems. This study not only demonstrates the potential of the proposed cache controller design in bridging the processor-memory speed gap but also contributes significantly to the field of cache memory management by offering a viable solution to the challenges posed by traditional cache configurations.

Copyright © 2023 Institute of Advanced Engineering and Science. All rights reserved.

Corresponding Author:

Teddy Surya Gunawan
Department of Electrical and Computer Engineering,
International Islamic University Malaysia, Malaysia.
Email: tsgunawan@iium.edu.my

1. INTRODUCTION

Cache memory is an integral component of modern computing systems, bridging the speed gap between fast processors and slow main memory, especially given Moore's Law's exponential increase in processing power. Even though the number of transistors doubles approximately every two years, main memory speeds have not kept pace, resulting in potential performance bottlenecks [1, 2]. Cache memory functions as a high-speed buffer to counteract this, enabling processors to retrieve data quickly. The cache controller is essential for optimizing the performance of the cache memory. This vital component, directed by the Finite State Machine (FSM), coordinates the data flow and operations between the processor, main memory,

and cache. A competent FSM is essential to determine the optimal cache operations and ensure peak performance.

Moore's law graph demonstrates that processor speed has increased exponentially in recent years, outpacing memory performance [3, 4]. In contrast to memory performance, CPU speed increases exponentially. This disparity causes significant main memory access latency in modern processors, and this trend is likely to persist. Cache memories are designed to reduce this latency. It is essential to improve cache memory systems to match the rapid increase in processor performance, with the design of efficient cache controllers being one effective strategy.

There are two essential terms in cache memory: hit and miss. A cache hit occurs when the CPU locates the requested data in the cache memory, whereas a cache miss indicates that the required data is absent [5-7]. Understanding these concepts is crucial to mastering the core cache memory operations: read and write. Delving deeper, we identify four distinct scenarios within these operations: read hit, read miss, write hit, and write miss.

The dependency of cache memory on the "Principle of Locality" is highlighted in [8]. Essentially, cache memory repeatedly and rapidly accesses a set of memory addresses. This behavior is consistent with the observation that a processor frequently revisits and reuses nearby memory addresses rather than fetching new data continuously [9]. The Principle of Locality has two components: temporal locality and spatial locality. According to the temporal locality, recently accessed data will likely be required again soon, while spatial locality asserts that data near a recently accessed address will also likely be accessed. Given the disparity between cache lines and main memory blocks, an algorithm is required to address the mapping problem. The architecture of cache memory is dependent on the mapping function selected. Address mapping schemes are essential cache parameters [10, 11] that determine where higher-level memory entries are stored in the cache. Set-associative, fully associative, and direct are the three predominant mapping techniques.

The design and assessment of a four-way set associative cache memory controller was investigated in [3]. The controller, essential for streamlining data transfers between the CPU, cache, and main memory, promises to reduce processor-memory delays significantly because processor data-handling capabilities are advancing more rapidly than memory speeds. A finite state machine implements each of the four operations. Using ISE Design Suite and Cadence compilers, the study analyzed the functionality of various modules, including evaluating "hit" and "miss" conditions. In tests, there were 19 successes and 6 failures, with the failures primarily attributable to specific instructions. Technical specifications for the controller included a setup time of 1.66 ns, an output time of 0.77 ns, a clock frequency of 257.202 MHz, and a modest power consumption of 5.53 mW.

The design of an efficient cache controller is investigated in [5] by comparing Direct mapped and 4-way Set associative mapped cache mappings. The controller was created with VHDL and simulated with Ed Playground. The architecture provided multiple input and output signals. Interacting with the CPU, reading, and writing to main and cache memory, and delivering the requested block to the CPU are the core functions of the cache controller. Test vectors covering Read and Write hits and misses were applied to both mappings to validate the design. The 4-way set associative mapping uses the Tree PLRU algorithm, which improves the fundamental LRU algorithms. According to comparative results, 4-way set associative mapping is more effective than direct mapping. The direct-mapped controller requires 2 cycles for a successful read, 6 cycles for a failed read, and 4 cycles for write operations. In contrast, the 4-way set associative controller requires three cycles for a read success, seven cycles for a read failure, and five cycles for write operations.

A power-efficient FPGA-based cache memory system was developed in [12] to detect cache memory miss rates. They highlighted the energy efficiency and performance benefits of FPGA-based computing. The cache memory design includes a counter, a cache tag comparator, and a cache tag memory but omits the cache data memory, as the emphasis is on cache miss detection. These components were created using the behavioral modeling technique of VHDL and synthesized on the Xilinx platform. If the microprocessor's requested address matches the one in the cache tag memory, a cache hit occurs; otherwise, a cache miss occurs. The study's conclusion presented the FPGA-based cache memory design for cache miss detection, which could lay the foundation for future FPGA-based set associative cache memory and controller projects.

Modern systems require higher CPU clock speeds, which increases power consumption and heat. Thus, multicore processors are recommended over high-frequency single-core ones. Cache coherence is highlighted in the cache controller architecture for multicore processors. Cache coherence was defined as core-wide memory reads and writes. A unified memory view is maintained when one core writes to a memory location and another reads from it. The three-core processor cache controller mediates between cache memory and CPU cores using four bus types for write, read, data, and address signals. Scheduling allows each core to access shared components like buses and memory in a pre-defined sequence to maintain cache coherence. Simulations in [13] showed that scheduling ensures that each core fetches the correct data according to the cache controller's timetable.

The performance of a cache memory system is highly dependent on the effectiveness of the cache controller, which is primarily determined by its architectural design. The mapping function is the foundation of this design. The simplest form of mapping, direct mapping, assigns a unique main memory block to a specific cache line. However, when multiple memory blocks compete for the same cache space, this singular mapping can result in a high cache miss rate. In contrast, fully associative mapping, which necessitates searching all cache lines for the desired data, can be time-consuming and detrimental to performance. An intermediate solution set associative mapping is not devoid of difficulties: excessive ways complicate replacement policies, while insufficient ways increase the rate of misses. The existing research does not comprehensively understand designing a VHDL 4-way set associative cache controller. Therefore, this research aims to design an FSM-based 4-way set associative cache controller using VHDL, verify its behavior through Quartus Lite Edition's timing diagram, and evaluate its performance via ModelSim simulation.

2. CACHE MEMORY PRINCIPLES

This section explores the fundamental principles of cache memory, including its complex mapping functions, the nuances of replacement algorithms and writes policies, and the cache controller's central role in the overall system.

2.1. Cache Mapping Functions

Mapping main memory blocks to cache lines is a complex task, necessitating specific algorithms, especially given the limited cache lines compared to main memory blocks. The architecture of the cache is influenced by the selected mapping function [14, 15]. Address mapping schemes were emphasized in [10] as pivotal cache parameters for this allocation. Three primary mapping techniques are detailed: direct mapping, fully associative, and set-associative. In direct mapping, each main memory block is mapped to only one specific cache line, determined by the address of the block multiplied by its modulo. This method is straightforward, as it doesn't require a replacement process; however, its rigidity may cause frequent block swaps in the cache if a program accesses different blocks mapping to the same line, leading to "trashing." Fully associative mapping is more versatile than direct mapping, allowing block data to be copied to any cache line. When no lines are free, a replacement algorithm comes into play.

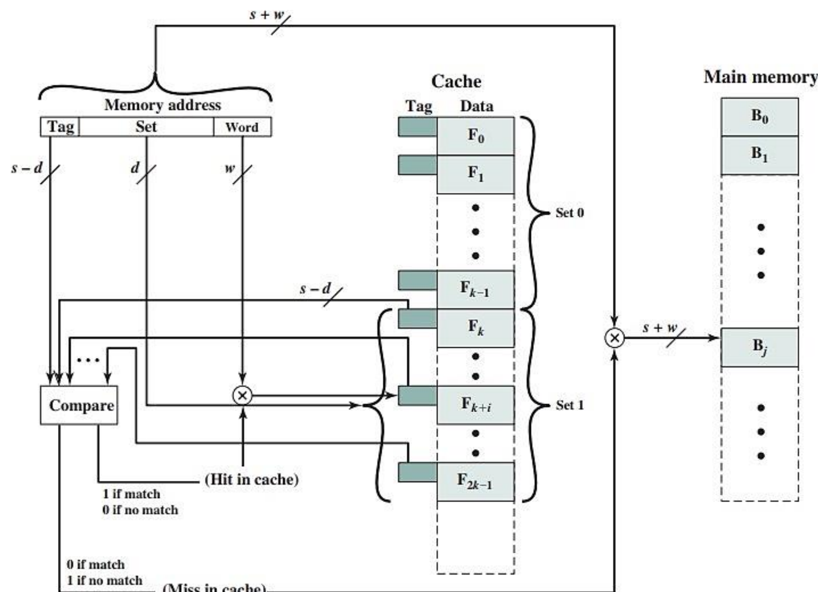

Figure 1. The *k*-way Set-Associative Cache Organization [16]

We have set associative mapping by combining the advantages of both the direct and associative techniques while minimizing their cons. It involves dividing all cache lines into sets based on a predetermined value *k*. While a specific block can be mapped to any line within a particular set, it's restricted to that set, as shown in Figure 1. Cache replacement algorithms might also be employed here if no lines are available. This method interprets a memory address via three fields: Tag, Set, and Word. The cache controller identifies the set through the Set field, compares each line's tag within the set, and then uses the Word field for line-byte selection. Notably, the set-associative approach can become direct mapping or fully associative based on the number of sets and lines in each set. Two-line set-associative organizations are typically the most common, with four-way set associative offering marginal improvements at a slightly added cost.

2.2. Replacement Algorithms and Write Policy

Cache memory design relies on the replacement algorithm [17]. This algorithm selects the cache line to replace with the main memory block. LFU, FIFO, and LRU are popular algorithms in this area (LRU). LRU stands out due to its intuitive nature and the underlying principle that recent memory accesses indicate near-future accesses, potentially optimizing hit ratios. LRU is described as replacing the cache line least recently accessed [18]. Several replacement algorithms will be discussed in this section.

First-In-First-Out (FIFO) is one of the simplest cache replacement algorithms. It operates on the age principle, replacing the oldest item in the cache when a new item needs to be loaded. The algorithm assumes that data brought into the cache earlier is less likely to be used again soon. Assuming a queue structure for the cache with items $C_1, C_2, \dots, C_n$, where C_1 is the oldest and C_n is the most recent, FIFO algorithm can be defined as:

$$FIFO(C) = \min\{in_time(C_i) | i = 1 \dots n\} \quad (1)$$

where $in_time(C_i)$ is the time when the item C_i entered the cache.

Random Replacement does exactly as its name suggests: it replaces a randomly chosen item in the cache when a new item is to be loaded. Due to its unpredictable nature, it might outperform more complex algorithms in certain scenarios, especially when access patterns do not exhibit strong temporal locality. The algorithm can be formulated as follows:

$$Random(C) = rand(\quad)\ mod\ n \quad (2)$$

where n is the cache size.

The **Optimal Replacement** algorithm is theoretical and serves as a benchmark. It replaces the item that won't be accessed for the longest duration in the future. While it's not feasible in practice (as it requires knowledge of future requests), it sets an upper-performance limit. The algorithm can be formulated as follows:

$$Optimal(C) = \max\{next_use(C_i) | i = 1 \dots n\} \quad (3)$$

where $next_use(C_i)$ denotes the future time when the item C_i will be accessed next.

Least Recently Used (LRU) is based on the principle that if an item has not been accessed recently, it's less likely to be accessed shortly. It replaces the least recently accessed item when a new item is loaded. It can be defined as follows:

$$LRU(C) = \min\{last_access(C_i) | i = 1 \dots n\} \quad (4)$$

where $last_access(C_i)$ is the time when the item C_i was last accessed.

Tree Pseudo-Least Recently Used (PLRU) approximates the LRU policy commonly used for set-associative caches. Instead of perfectly keeping track of the exact LRU order, which can be hardware-intensive, Tree PLRU uses a binary tree to guide replacement decisions [19, 20]. It makes it more efficient in terms of hardware implementation for larger set-associative caches. Considering a cache with 2^m lines, PLRU utilizes a binary tree with 2^m leaves representing cache lines. Cache accesses alter the state of the tree. An internal node's state, set to '0' or '1', indicates which child node (or associated cache line) was accessed last. The algorithm determines the least recently used line by following the path from the root according to these states.

When updating a cache-resident block, there are two possible outcomes. First, if the block is unchanged, it can be overwritten in the cache without updating the main memory. Before introducing a new block, the main memory must be updated if the existing block has undergone at least one write operation. Diverse writing policies address these scenarios, each with its advantages and disadvantages. This dynamic reveals two principal concerns. First, multiple devices, such as I/O modules, may access the main memory, rendering cache-only modified words potentially invalid. This situation becomes more complex with multiple CPUs on a shared bus, each with its cache. Changing a cached word can invalidate words in other caches.

The "write-through" technique is simple. It ensures that both the cache and the main memory are updated simultaneously, preserving the integrity of the memory. Additionally, it enables other processor-cache modules to monitor main memory traffic, ensuring cache consistency. In contrast, the "write-back" technique uses additional flags in each cache line, such as Type (differentiating between data and instructions), Valid (indicating valid cache line data), Lock (preventing line replacement when set), and Dirty (denoting data written to cache but not main memory). As explained in [21], the write-back policy minimizes memory writes by updating the main memory only when the dirty bit is active during block replacement. The write-back policy

struggles with error handling without an Error Correcting Code (ECC), and the write-through approach can excessively congest traffic [8, 22].

2.2. Cache Controller

The cache controller is an essential piece of hardware that manages data transfers between the CPU, main memory, and cache memory. Whenever the processor attempts to access a specific location in the main memory, the cache controller first determines whether the data is present in the cache. If present, the data is transmitted directly to the processor; otherwise, the data is retrieved from the main memory, thereby updating the cache [3]. In addition, the cache controller is responsible for monitoring the cache's induced miss rate [12, 23].

The cache controller comprises a Finite State Machine (FSM), Tag cache, General Mux (GMux), and the LRU controller unit. The FSM manages read and write operations for cache and main memory, directing requests to GMux and the LRU controller unit for set associativity [18, 24]. The tag cache organizes data into distinct fields within the controller, including tag bits, a valid bit, a dirty bit, and LRU bits for each data cache line. The FSM generates signals such as "Dwith," making it easier for GMux to connect input and output data buses. When set associative mode is enabled, the LRU controller identifies the path utilized the least recently while the FSM manages read and write operations for each set.

3. PROPOSED FAST 4-WAY SET ASSOCIATIVE CACHE CONTROLLER

This research focuses on modeling the cache controller to validate its functionality and simulate its performance. This modeling utilizes HDL because it can represent behavior at multiple abstraction levels. Abstraction is a pillar of engineering; it facilitates the comprehension of system operations without delving into complex internal processes [25]. In addition, omitting the specifics of the low-level implementation ensures that the simulation can be executed within a reasonable timeframe. Our research uses the Register Transfer Level, where HDL specifies the transfer and transformation of data across and within subsystems in response to system inputs.

3.1. Cache Controller and Cache Memory Specifications

Before designing the cache controller, it is necessary to define its inputs and outputs, as shown in Table 1. This project uses a 4-way set associative mapping for cache memory, which consists of 16 cache lines of 256 bytes each. This memory is divided into four sets, each containing four cache lines. Write-Through for write hits and Write-Around for write misses are the chosen write policies. Adopting the Tree Pseudo-LRU, the research introduces improvements to the replacement policy (PLRU). In addition, the cache controller includes a Tag Array for storing tags for comparison with the processor's provided address. Figure 2 demonstrates the complete RTL layout. Standard bus specifications include 32-bit data buses, a 32-bit address bus (with only 10 bits addressable by memory), and a 128-bit read data block.

Table 1. I/O Ports for the Cache Controller

Signal	Direction	Width	Description
Clock	In	1	Global clock
Reset	In	1	Async. Reset
Flush	In	1	To flush all cache lines
Read	In	1	To read words from the cache
Write	In	1	To write words to cache
Index	In	Configurable	Index of the requested address
Tag	In	Configurable	Tag of the requested address
Ready	In	1	The data-ready signal from the main memory
Refill	Out	1	To refill the cache line from the main memory
Set offset	Out	2	To select one of the four ways
Update	Out	1	To update cache line using processor word, on write hit
Stall	Out	1	To stall the processor on read/write miss, write hit
Read from memory	Out	1	To fetch cache line from main memory, on read miss
Write to memory	Out	1	To write processor word to main memory

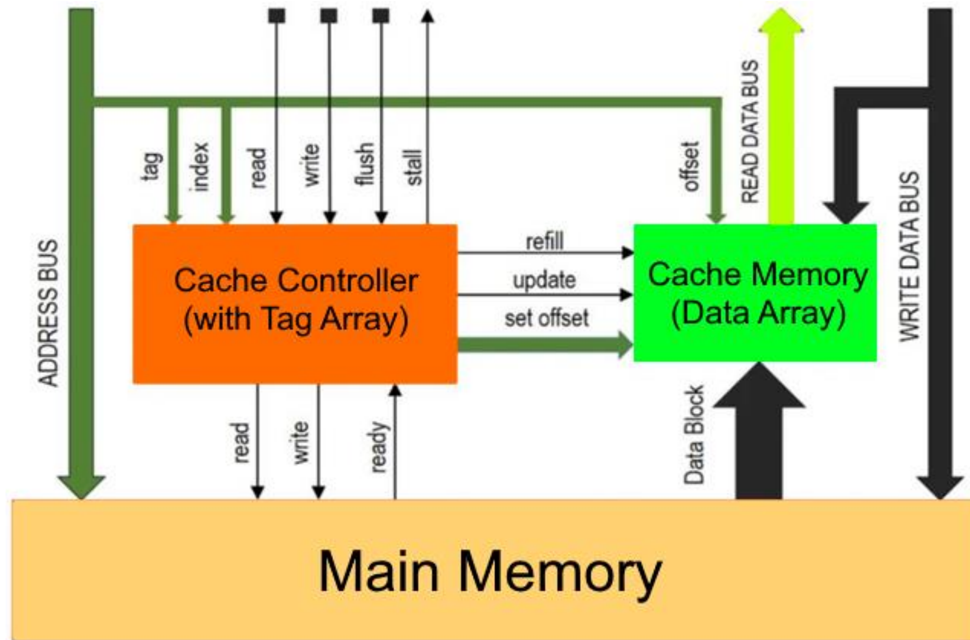

Figure 2. RTL View of the Entire System

3.2. System Design Using Finite State Machine

After understanding the cache controller and memory specifications, the next step is to design the Finite State Machine (FSM). FSMs are among the most powerful digital circuits due to their capacity to represent and manage distinct system states. In the context of this study, the FSM allows us to precisely define the behavior of the cache controller based on its inputs. The architecture of our cache controller is based on prior research and consists of an FSM that performs four fundamental operations: retrieving addresses from the processor, reading data from both cache and main memory, writing data to cache and main memory, and finally returning the requested data to the processor. Figure 3 depicts a representative state diagram of the cache controller.

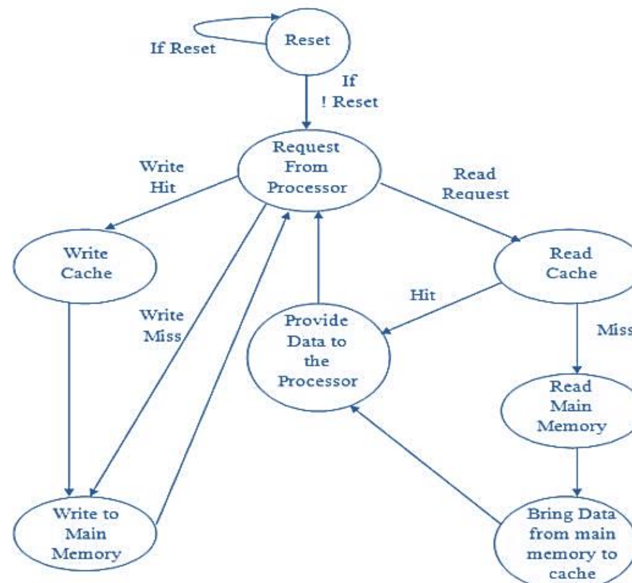

Figure 3. Cache Controller State Diagram

3.3. Integrated System Design and Development

Using Quartus Lite software, the intended circuit is constructed textually, utilizing the fundamental Finite State Machine (FSM) to streamline VHDL design into distinct procedural blocks for different circuits. Quartus transforms VHDL code into a tangible circuit with logic elements, providing a schematic

representation and preparing it for FPGA implementation, upon compilation. A subsequent step entails the creation of a Test Bench, which is specific to the module code, to direct the simulator, ModelSim, to evaluate the operational efficiency of the circuit without delving into timing nuances. Timing remains crucial, with clock stall metrics highlighting the model's performance; extensive clock stalls indicate potential processor data retrieval delays. Due to Quartus' affiliation with Intel, those with the necessary hardware can implement the model on an Intel FPGA device, with the final phase focusing on the programming required to finalize the system's architecture.

3.3. Netlist View

We will investigate the complexities of our circuit's netlist. This netlist, which serves as a blueprint for the interconnectedness of the system's components, is the result of successful code synthesis and compilation. The system's architecture consists of three essential components: the cache controller, the cache memory, and the main memory. Figure 4 vividly depicts the meticulous integration of individual entities to form the overarching "top entity" in constructing such a system. Each entity has its own unique significance and design considerations.

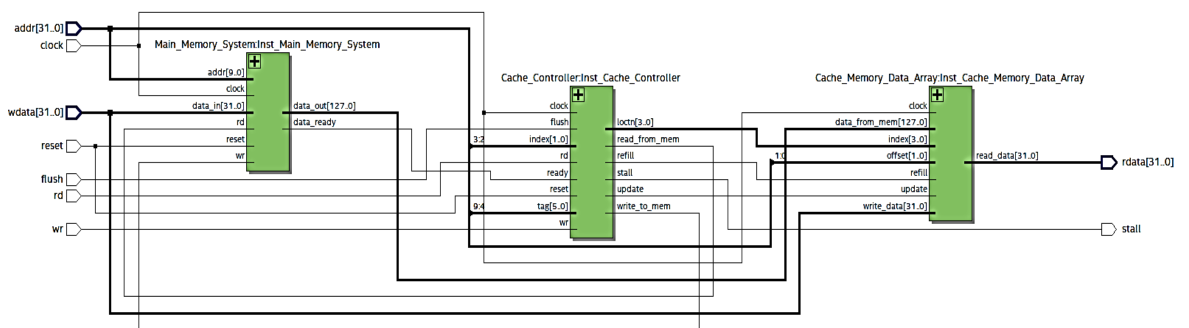

Figure 4. Top Entity of the System

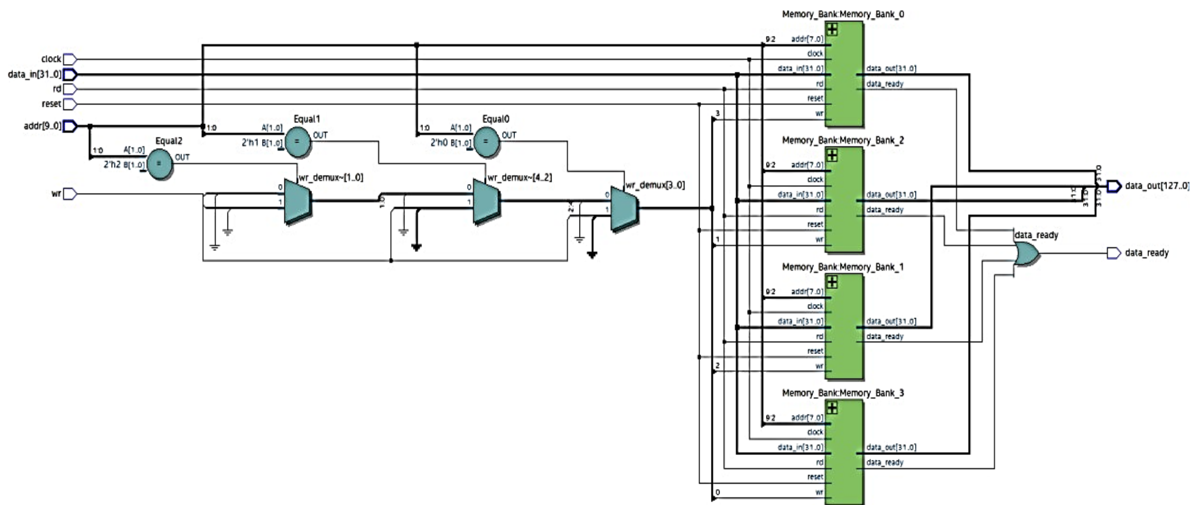

Figure 5. Main Memory Entity

The primary memory is our first entity. It is intricately designed and comprises four distinct banks, as shown in Figure 5. To address the reviewer's query, each memory bank is associated with a dedicated data_out bus. Specifically, **Memory_Bank_0** produces **data_out[31:0]**, **Memory_Bank_1** handles **data_out[63:32]**, **Memory_Bank_2** is responsible for **data_out[95:64]**, and **Memory_Bank_3** manages **data_out[127:96]**. This configuration ensures efficient data handling and output across the different segments of the memory system.

The focus then shifts to the second entity, the cache controller, which consists of two segments, as depicted in Figure 6. The cache memory, a crucial component of our system, effectively interfaces with these memory banks to optimize data retrieval and storage processes.

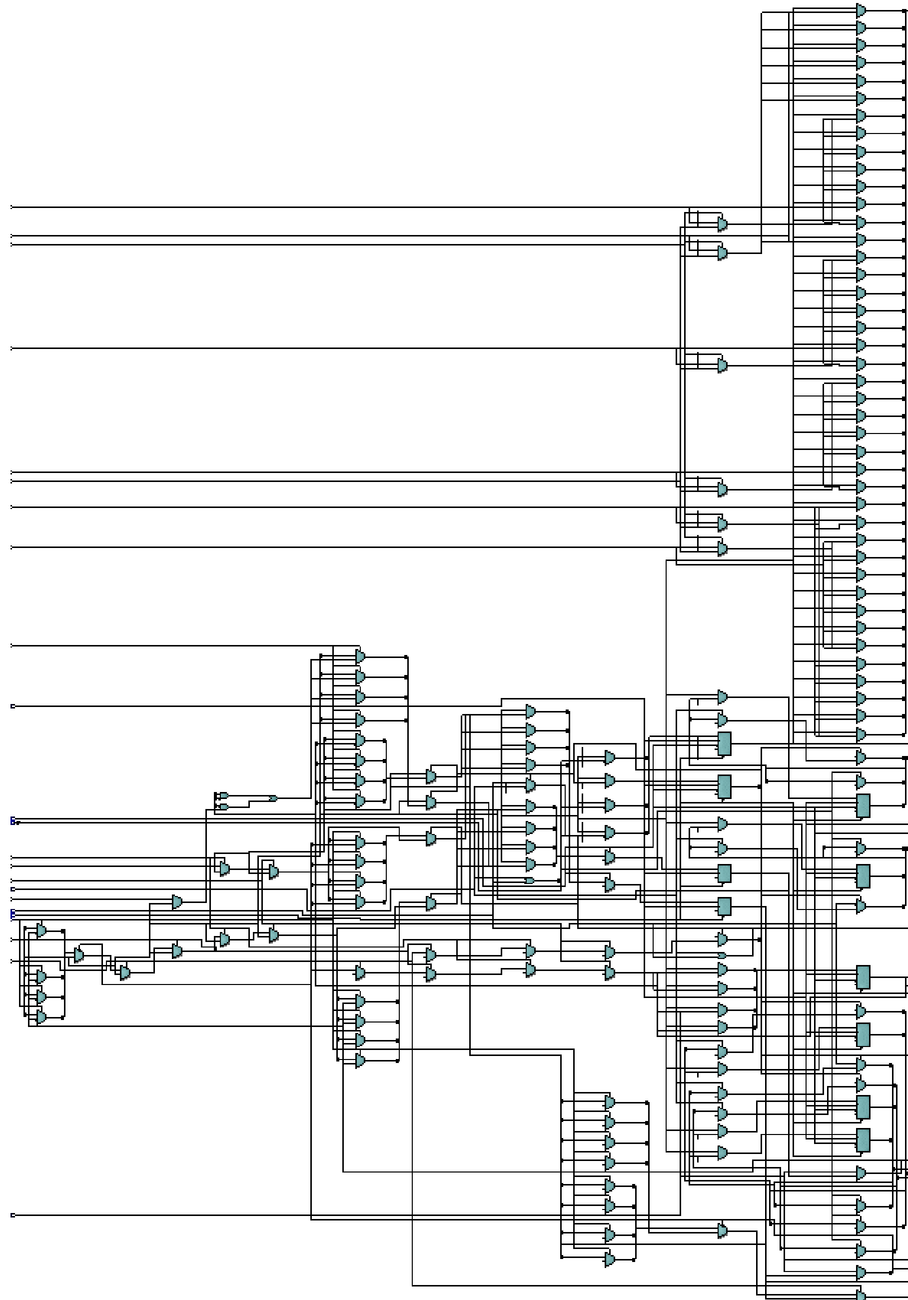

Figure 6. Cache Controller Entity

4. RESULTS AND DISCUSSION

This section explores the fundamental principles of cache memory, including its complex mapping functions, the nuances of replacement algorithms and writes policies, and the cache controller's central role in the overall system. In our examination of the performance of the 'Top entity,' we simulated the functions of a non-pipelined processor using a Test Bench. This Test Bench generated Read/Write data requests to the memory consistently, effectively simulating the "Load" and "Store" instructions inherent to all programs a processor executes. Our tests revealed that the cache controller was operating at peak efficiency. During the research, several notable observations were made. First, the Read/Write Miss and Hit signals were generated with extraordinary accuracy. Second, all four routes for the Read/Write data operations were fully functional. In addition, the pLRU algorithm demonstrated its proficiency by replacing cache lines as anticipated. Notably, we did not encounter incoherent or inconsistent data during our research. To ensure a thorough evaluation, we administered three sets of test vectors, one for each scenario: read hit, read miss, write hit, and write miss. The subsequent section will comprehensively analyze these scenarios, spanning all three sets.

4.1. First Set of Test Vector Analysis

The initial set of test vectors mirrored those of a previously cited study to facilitate a direct comparison with prior research. The overarching objective was to identify potential inconsistencies or flaws in this or the other study. The results showed:

- **Read Miss**: For addresses not present in the cache, the data block is fetched from the main memory, stalling the processor for 3 cycles and taking a total of 5.5 cycles for the read operation.
- **Read Hit**: If the address is in the cache, data is directly fetched, requiring 3 cycles.
- **Write Hit**: When the target address is in the cache, data is written and simultaneously copied to the main memory, owing to the Write-Through policy. This operation takes 3.5 cycles, with a 2-cycle processor stall.

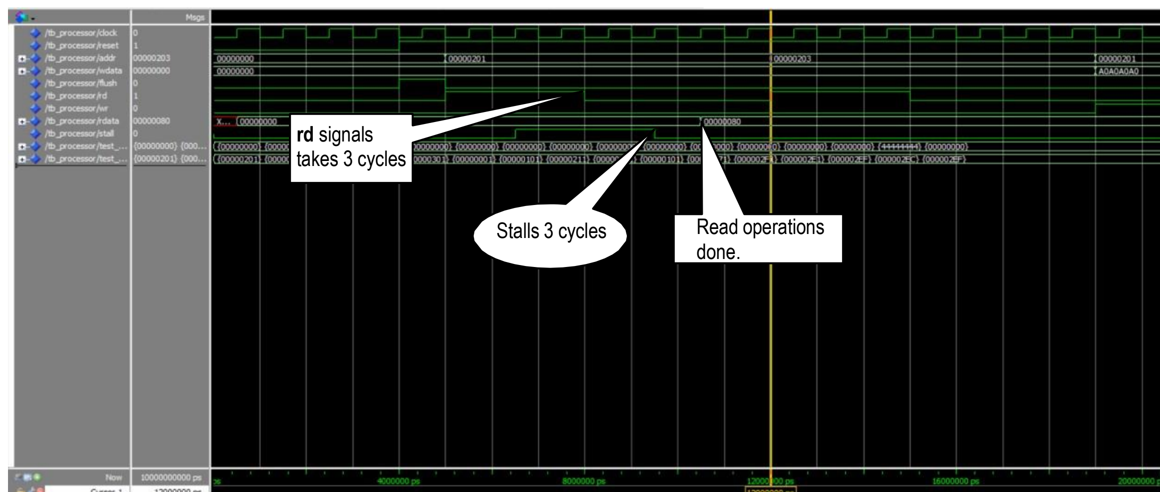

Figure 7. Simulation result of **test_addr(0)=00000201** for the read miss.

Figure 7 zeroes in on the specific instance of **test_addr(0) <= x "00000201"**, marking a situation where a read miss is registered. The processor's incoming address is mirrored in the **tb_processor/addr** signal. Notably, the **rd signal**, set at '1', indicates a read operation that, intriguingly, consumes 3 cycles. The dilemma arises, however, when the desired address is conspicuously absent from the cache. It triggers the transfer of an entire data block from the main memory to the cache before sending it to the processor. While effective at ensuring data integrity, this process introduces a delay: the cache controller temporarily holds the processor in check for three additional cycles. This results in a total of 5.5 cycles for read operations should a miss occur. It highlights the inherent impact of cache misses on system performance and the significance of optimizing cache management to reduce their occurrence.

4.2. Second Set of Test Vector Analysis

This set introduced the 'Write Miss' operation, which was not covered in the previous set. The analysis showed:

- **Read Miss**: Akin to the first set, data retrieval from the main memory resulted in a 3-cycle processor stall, with the read operation consuming 4.5 cycles.
- **Read Hit**: Consistent with the first set, direct data retrieval from the cache took 3 cycles.
- **Write Miss**: Direct writing to the main memory occurred when the desired address wasn't in the cache. This process took 3.5 cycles, stalling the processor for 2 cycles.
- **Write Hit**: Similar to the first set, a 3.5-cycle operation was observed with data being written in the cache and copied to the main memory, resulting in a 2-cycle processor stall.

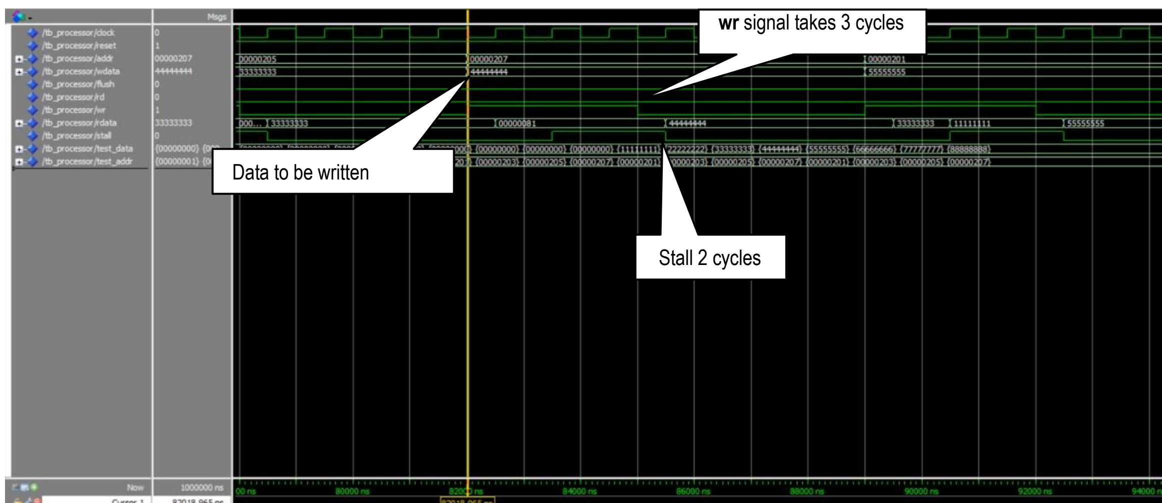

Figure 8. Simulation result of **test_addr (11) =00000207** for the write miss

Figure 8 elucidates a scenario involving a write miss, characterized by the specific instance **test_addr (11) <= x "00000207"**. The crux of this situation lies in the absence of the desired address within the cache memory, directly categorizing the write request as a miss. Instead of leveraging the cache, data is straightforwardly written to the main memory, denoted by **test_data (11) <= "44444444"**. A critical observation here revolves around the **wr signal**, set to '1'. This signal denotes a write operation, which requires three cycles as deduced from the results. Despite being direct, this operational procedure introduces a latent overhead: the processor is temporarily suspended for an additional 2 cycles. The cumulative effect of write misses is an access time of 3.5 cycles. It elucidates the inherent overhead of write misses and highlights the urgent need to optimize cache strategies to prevent such occurrences and boost overall system efficiency.

4.3. Third Set of Test Vector Analysis

The third set provided a concise overview of the cache controller's behavior, allowing observation of specific scenarios like data updating on the same address. The analysis showed:

- **Read Miss and Hit**: Behaviors were consistent with previous sets.
- **Write Miss and Hit**: The cache controller responded correctly to write requests, showcasing its ability to handle data updates on the same address.

The third set of test vectors provides a more constrained coverage than the preceding first and second sets. The design's rationale is an essential factor to consider in this context. Although limited in scope, this set is intricately designed to examine the behavior of the cache controller in four distinct scenarios: read miss, read hit, write miss, and write hit, with the added complexity of data updates on identical addresses. This deliberate limitation suggests a strong emphasis on depth rather than breadth. The value of such a strategy is evident: by narrowing the scope, there is increased observational granularity. It allows for a more in-depth understanding of the cache controller's operations in particular scenarios instead of a broader but potentially superficial understanding.

Figure 9 provides additional visual distinction. Each scenario is color-coded distinctly: blue represents read misses, red represents read hits, green represents write misses, and orange represents write hits. Although this color coding is intuitive, one may question whether these choices effectively accommodate all potential viewers, including those with color blindness.

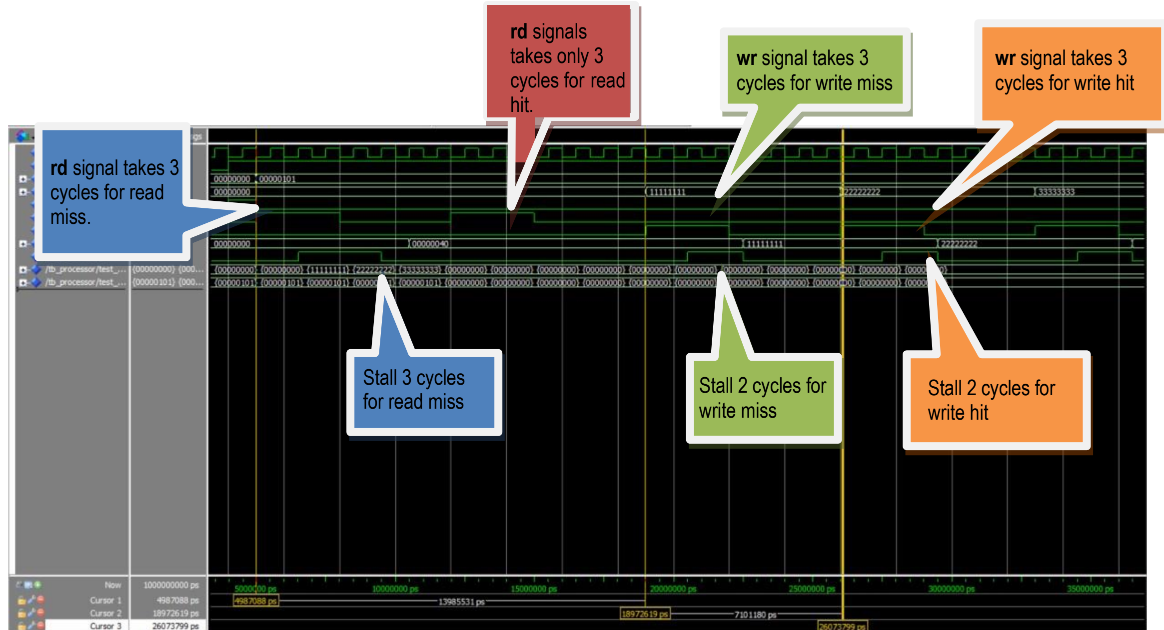

Figure 9. Simulation result for the third set of test vector

Reading hits are the most effective operations, requiring only three cycles and causing no processor delays. In contrast, reading misses require data retrieval from the main memory, resulting in a 5.5-cycle operation. Both write hits and misses presented similar difficulties, halting the processor for two cycles and requiring three and a half cycles to gain access. The Write-Through policy of the design ensured data consistency by writing to the main memory with each writing hit. In addition, the Write-around policy became evident during writing misses, wherein data was written directly to the main memory without updating the cache.

Table 2. Comparison with other works

Ref	Read Hit (cycles)	Read Miss (cycles)	Write Hit (cycles)	Write Miss (Cycles)
[3]	4	9	3	2
[5]	3	7	5	5
Proposed design	3	5.5	3.5	3.5

Table 2 compares our design with the other works, revealing that the cycle efficiency of cache interactions has evolved. A relatively unbalanced cycle distribution is evident in [3], with read misses dominating by a significant margin at 9 cycles, nearly double its nearest metric (read hits at 4 cycles). It could indicate a system that, while robust, may encounter difficulties in scenarios involving frequent data access. In contrast, a more balanced cycle distribution was displayed in [5], particularly between write operations. The reduced cycle count for read misses compared to [3] (7 cycles) suggests read efficiency-enhancing optimizations. However, it also features a significantly higher cycle count for write operations, which may indicate a potential design trade-off.

The proposed design provides convincing evidence of a refined approach. With a consistent cycle count for both read and write hits of 3 and 3.5 cycles, it establishes a balance between read and write operations, thereby addressing potential inefficiencies in both. Moreover, the significantly reduced cycle count for read misses to 5.5 cycles demonstrates a read mechanism that has been optimized, most likely utilizing innovative techniques or algorithms. In essence, the proposed design appears to have integrated the lessons from the two previous papers, combining their respective strengths to create a system that promises speed and balanced performance across diverse cache interactions.

Beyond the comparative analysis outlined in Table 2, it is critical to contemplate the ramifications that the implementation of FPGA would have on our proposed design. The present investigation is founded upon ModelSim simulations; however, it is expected that the adoption of an FPGA platform, such as the Cyclone V FPGA, will verify and potentially augment these findings. The utilization of FPGA implementation provides a concrete setting in which the design can be evaluated under authentic circumstances, thereby yielding significant knowledge regarding timing, power consumption, and system integration as a whole.

The efficacy of our simulation is demonstrated by the cycle efficiency outcomes, which indicate that it accurately models the behavior of the cache controller in a variety of scenarios. By providing a controlled

environment in which to precisely manipulate each parameter, simulations guarantee a comprehensive comprehension of the performance of the design. By employing this comprehensive simulation methodology, a strong groundwork is established for subsequent FPGA implementation, during which the design's practical viability will be further evaluated. Furthermore, it is anticipated that the shift from simulation to FPGA implementation will provide further advantages. For example, testing on FPGAs can provide valuable information regarding the scalability and adaptability of the design across various hardware configurations. Additionally, it facilitates debugging and testing in real-time, which can result in more immediate and tangible design enhancements. Hence, although the present investigation centers on performance analysis via simulation, the potential implementation of our cache controller design on an FPGA is an essential subsequent course of action. In addition to validating the results obtained from our simulations, it will furnish an all-encompassing comprehension of the design's operational efficacy. By integrating comprehensive simulations and rigorous hardware testing, our proposed design is characterized by its exceptional dependability and groundbreaking nature.

5. CONCLUSIONS

The primary objective of this study was to develop and evaluate a cache controller that utilized a Tree Pseudo Least Recently Used (PLRU) replacement policy and a 4-way set associative mapped cache. An exhaustive literature review was conducted to identify effective cache controller design techniques; this led to the selection of a four-way set for each cache set, which balanced complexity, performance, and cost while optimizing latency. During the practical implementation phase, Quartus Prime 16.1 Lite Edition was utilized to develop VHDL code for the cache controller, main memory system, and cache memory entities. The conversion of abstract principles into a physical design during this critical stage was accomplished with the assistance of compilation, synthesis, and ModelSim simulations for validation purposes. The timing diagrams generated by these simulations served as the foundation for our performance evaluation. Through comparative analysis with prior research, the effectiveness of our design was demonstrated, specifically in the reduction of read miss latency to 5.5 cycles and the attainment of balanced latencies in write operations. It is important to mention that the performance of the cache controller is affected by a multitude of factors, such as the specifications of the device and the replacement and writing policies that are chosen. Our investigation was limited to operation cycles; internal delay considerations were disregarded. In its entirety, this research makes a substantial contribution to the field of cache controller design by showcasing the efficacy and feasibility of the 4-way set associative mapped cache controller in conjunction with the Tree PLRU policy through simulations. Subsequent research endeavors to incorporate and scrutinize this architecture on an FPGA platform, thereby furnishing an exhaustive assessment of its practical efficacy. This stage is critical to verify the feasibility and efficacy of the design in real-world situations, thereby connecting theoretical analysis with practical implementation and contributing to the advancement of cache controller technology.

ACKNOWLEDGEMENTS

The authors are grateful to the International Islamic University Malaysia for its research facilities. They also thank Universitas Negeri Yogyakarta for their generous funding and facilities for this research.

REFERENCES

[1] M. Gupta, L. Bhargava, and S. Indu, "Mapping techniques in multicore processors: current and future trends," *The Journal of Supercomputing,* vol. 77, pp. 9308-9363, 2021.
[2] A. Farshin, A. Roozbeh, G. Q. Maguire Jr, and D. Kostić, "Make the most out of last level cache in intel processors," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1-17.
[3] P. Chauan, G. Singh, and G. Singh, "Cache controller for 4-way set-associative cache memory," *International Journal of Computer Applications,* vol. 129, no. 1, p. 8887, 2015.
[4] P. Visconti, R. Velazquez, C. D.-V. Soto, and R. De Fazio, "FPGA based technical solutions for high throughput data processing and encryption for 5G communication: A review," *TELKOMNIKA (Telecommunication Computing Electronics and Control),* vol. 19, no. 4, pp. 1291-1306, 2021.
[5] G. Kaur, R. Arora, and S. S. Panchal, "Implementation and Comparison of Direct mapped and 4-way Set Associative mapped Cache Controller in VHDL," in *2021 8th International Conference on Signal Processing and Integrated Networks (SPIN)*, 2021: IEEE, pp. 1018-1023.
[6] N. Beckmann, H. Chen, and A. Cidon, "{LHD}: Improving cache hit rate by maximizing hit density," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 389-403.
[7] E. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," in *International Conference on Machine Learning*, 2020: PMLR, pp. 6237-6247.
[8] I. Lokegaonkar, D. Nair, and V. Kulkarni, "Enhancement of cache memory performance," in *2021 3rd International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)*, 2021: IEEE, pp. 1490-1492.

[9] C. Griner, S. Schmid, and C. Avin, "CacheNet: Leveraging the principle of locality in reconfigurable network design," *Computer Networks,* vol. 204, p. 108648, 2022.

[10] M. Jhamb, R. Sharma, and A. Gupta, "A high level implementation and performance evaluation of level-I asynchronous cache on FPGA," *Journal of King Saud University-Computer and Information Sciences,* vol. 29, no. 3, pp. 410-425, 2017.

[11] M. R. Khalil, L. A. Mohammed, and O. N. Yousif, "Customer application protocol for data transfer between embedded processor and microcontroller systems," *TELKOMNIKA (Telecommunication Computing Electronics and Control),* vol. 19, no. 3, pp. 801-808, 2021.

[12] Y. S. Watile and A. Khobragade, "FPGA Implementation of cache memory," *International Journal of Engineering Research and Applications (IJERA),* vol. 3, no. 3, pp. 283-286, 2013.

[13] V. S. Bhure and P. R. Chakole, "Design of cache controller for multicore processor system," *International Journal of Electronics and Computer Science Engineering,* 2012.

[14] S. Kumar and P. Singh, "An overview of modern cache memory and performance analysis of replacement policies," in *2016 IEEE International Conference on Engineering and Technology (ICETECH)*, 2016: IEEE, pp. 210-214.

[15] A. Alsharef, P. Jain, M. Arora, S. R. Zahra, and G. Gupta, "Cache memory: an analysis on performance issues," in *2021 8th international conference on computing for sustainable global development (INDIACom)*, 2021: IEEE, pp. 184-188.

[16] W. Stallings, *Computer Organization and Architecture: Designing for Performance*. Pearson, 2018.

[17] M. T. Banday and M. Khan, "A study of recent advances in cache memories," in *2014 International Conference on Contemporary Computing and Informatics (IC3I)*, 2014: IEEE, pp. 398-403.

[18] S. S. Omran and I. A. Amory, "Design of two dimensional reconfigurable cache memory using FPGA," in *2016 5th International Conference on Electronic Devices, Systems and Applications (ICEDSA)*, 2016: IEEE, pp. 1-8.

[19] D. Grund and J. Reineke, "Toward precise PLRU cache analysis," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, 2010: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[20] B. U. I. Khan, R. F. Olanrewaju, R. N. Mir, A. R. Khan, and S. Yusoff, "A Computationally Efficient P-LRU based Optimal Cache Heap Object Replacement Policy," *International Journal of Advanced Computer Science and Applications,* vol. 8, no. 1, 2017.

[21] H. S. Mahmood and S. S. Omran, "Pipelined MIPS processor with cache controller using VHDL implementation for educational purposes," in *2013 International Conference on Electrical Communication, Computer, Power, and Control Engineering (ICECCPCE)*, 2013: IEEE, pp. 82-87.

[22] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu, "Adaptive cache management for energy-efficient GPU computing," in *2014 47th Annual IEEE/ACM international symposium on microarchitecture*, 2014: IEEE, pp. 343-355.

[23] S. Srivastava and P. Singh, "HCIP: Hybrid Short Long History Table-based Cache Instruction Prefetcher," *International Journal of Next-Generation Computing,* vol. 13, no. 3, 2022.

[24] B. Kumar, A. K. Bhosale, M. Fujita, and V. Singh, "Validating multi-processor cache coherence mechanisms under diminished observability," in *2019 IEEE 28th Asian Test Symposium (ATS)*, 2019: IEEE, pp. 99-995.

[25] B. J. LaMeres, *Quick Start Guide to Verilog*. Springer, 2019.