Design and Implementation of a Memory Management Unit and Cache Controller in RTL

(Monsoon 2025)

Submitted by

Kalrav Mathur (2210110338)

Akhil Sriram (2210110134)

Under Supervision
of

Dr. Venkatnarayan Hariharan
Department of Electrical Engineering

Abstract

This project presents the complete Register Transfer Level (RTL) design, implementation, and verification of a memory management subsystem, comprising a Translation Lookaside Buffer (TLB)-based Memory Management Unit (MMU) and a 2-Way Set-Associative Cache Controller. The system is engineered to address the "memory wall" bottleneck by providing high-speed virtual-to-physical address translation and efficient data caching. The design is implemented in Verilog-2001, emphasizing modularity to support future multi-level hierarchies. Key contributions include a robust Finite State Machine (FSM) for cache control, an invalidation-based coherence mechanism for Write-Through policies, and a novel script-driven verification environment. This environment parses human-readable instruction files (e.g., `R 0x1000`) to drive simulation scenarios, bridging the gap between architectural intent and RTL verification. Extensive simulations validate the correctness of the LRU replacement policy, hit/miss logic, and pipeline stall mechanisms under various load conditions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In modern computing, the performance gap between high-frequency processors and high-latency main memory (DRAM) necessitates sophisticated memory hierarchies. This project focuses on the hardware implementation of two critical components that bridge this gap: the Memory Management Unit (MMU) for virtual memory support and the Cache Controller for latency reduction.

The system is designed as a Physically Indexed, Physically Tagged (PIPT) cache, implying that address translation occurs prior to cache access. While this serializes the lookup path, it eliminates aliasing issues common in virtual caches, ensuring robust data integrity across processes.

1.1 Motivation

The implementation of a cache controller from scratch provides invaluable insights into micro-architectural trade-offs. By developing the RTL for tag comparison, victim selection, and write policies, we gain a cycle-accurate understanding of how hardware handles memory requests. Furthermore, the addition of a script-driven testbench allows for the rapid prototyping of complex access patterns, simulating real-world program behavior better than static test vectors.

1.2 Project Objectives

The core objective is to create a functional and modular RTL design of an MMU and a single-level cache controller. We aim to understand the intricate state machine logic required to handle various memory access scenarios, including cache hits, misses, and different write strategies.

1.2.1 Memory Management Unit (MMU) Scope

The MMU design focuses on address translation and protection. It implements a conceptual TLB model to map 32-bit virtual addresses to 32-bit physical addresses. Key functionalities include parsing the virtual address, performing a TLB lookup, and gener-

ating the corresponding physical address or signaling a miss to initiate a page table walk (simulated).

1.2.2 Cache Controller Scope

The cache controller manages an 8KB, 2-way set-associative cache. The design is parameterized to allow for configurable cache size, block size, and associativity. It supports a write-through policy with a no-write-allocate strategy for misses, utilizing a 1-bit Least Recently Used (LRU) policy for handling cache evictions.

1.3 Methodology

Our design methodology follows a rigorous RTL design flow tailored for ASIC implementation.

1. **Architectural Modeling:** Defining the state elements (SRAMs, Tag Stores) and control logic (FSM) based on theoretical models.
2. **RTL Design Entry:** Implementing the modules in Verilog-2001, ensuring synthesizability and modularity.
3. **Functional Verification:** Developing a comprehensive testbench environment using Synopsys VCS to validate functionality against corner cases.
4. **Debugging and Analysis:** Utilizing Synopsys Verdi to analyze waveforms, trace signal drivers, and verify FSM transitions.

1.3.1 Tools and Technologies

The project leverages industry-standard EDA tools for verification and analysis.

- **Synopsys VCS:** Used for high-performance compilation and simulation of the Verilog RTL.
- **Synopsys Verdi:** Employed for advanced waveform debugging and signal tracing.
- **Verilog-2001:** The primary hardware description language used for implementation.

Chapter 2

Literature Survey

2.1 Foundational Concepts in Computer Architecture

A review of foundational literature was conducted to establish the theoretical basis for our design. Textbooks such as "Computer Architecture: A Quantitative Approach" by Hennessy and Patterson and "Computer Architecture and Organization" by Smruti Sarangi provided comprehensive insights into memory hierarchies and caching principles [1].

For practical implementation strategies, we analyzed the open-source Bluespec implementation of the SHAKTI processor's cache controller [2]. Additionally, educational resources from GeeksforGeeks and Branch Education were consulted to reinforce concepts regarding VIPT caches and virtual memory mechanisms [3–6]. This research directly informed our decisions regarding the 2-way set-associative structure and the write-through policy implementation.

Table 2.1: System Configuration Parameters

Parameter	**Value**	**Parameter**	**Value**
Virtual Address Width	32 bits	Physical Address Width	32 bits
Cache Size	8 KB	Associativity	2-way
Block Size	64 Bytes	Write Policy	Write-Through
Page Size	4 KB	Replacement Policy	1-bit LRU

Chapter 3

Work Done

3.1 Project Progress

We have successfully completed the architectural design, RTL implementation, and functional verification of the entire memory subsystem. The following sections detail the specific implementations and the mathematical foundations governing the design.

3.1.1 System Architecture and Pipeline Design

The system comprises three main blocks: the CPU interface (simulated via testbench), the MMU, and the Cache Controller. The architecture follows a strict pipeline where the CPU issues a Virtual Address (VA), the MMU translates this to a Physical Address (PA) via the TLB, and the Cache Controller utilizes the PA to index into the SRAM arrays.

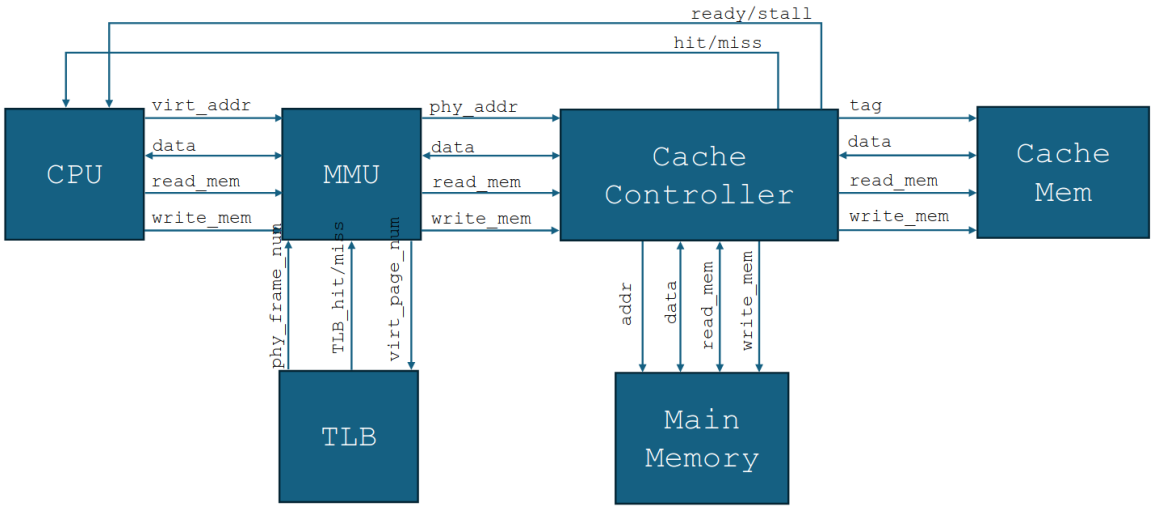


Figure 3.1: Overall System Architecture connecting CPU, MMU, Cache Controller, and Memory.

As shown in Figure 3.1, the CPU requests flow sequentially:

1. **CPU → MMU:** The CPU issues `virt_addr` along with `read_mem` or `write_mem` control signals.

2. **MMU → Cache Controller:** Upon a successful TLB hit, the MMU forwards the translated `phy_addr` to the Cache Controller.

3. **Cache Controller → Memory:** If a cache miss occurs, the controller communicates with Main Memory via the `addr` and `data` buses to fetch the block.

3.1.2 Memory Management Unit (MMU) Implementation

We implemented a fully associative TLB model within the MMU. Figure 3.2 illustrates the MMU's central role in translation.

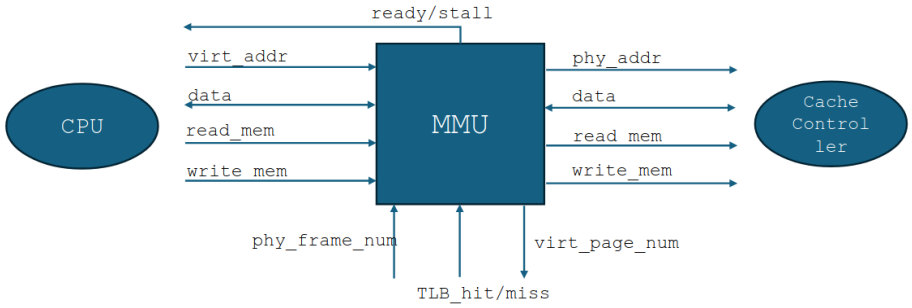


Figure 3.2: Memory Management Unit (MMU) Interface.

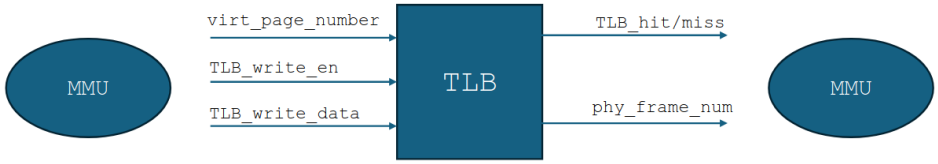


Figure 3.3: Translation Lookaside Buffer (TLB) Block Diagram.

The MMU integrates the TLB (Figure 3.3), which performs the critical lookup.

- **Lookup Logic:** The TLB compares the incoming `virt_page_number` against all valid tags simultaneously in a single clock cycle.

- **Refill Logic:** Upon a miss, the system signals a miss condition via `TLB_hit/miss`, which triggers a simulated Page Table Walk in the testbench.

3.1.3 Cache Controller Implementation

The Cache Controller is the most complex module, managing data flow and consistency. As depicted in Figure 3.4, it sits between the MMU, the Cache Memory (SRAM), and the CPU.

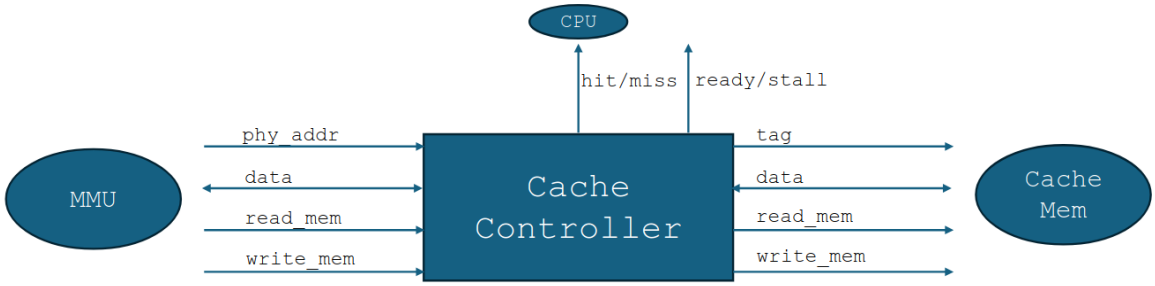
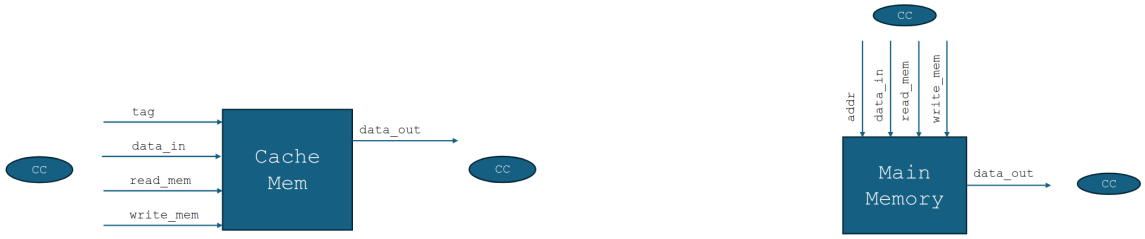


Figure 3.4: Cache Controller Interface Diagram.

The controller uses a robust Finite State Machine (FSM) to manage the following interfaces:

- **To CPU:** Provides `hit/miss` status and `ready/stall` signals to freeze the pipeline during misses.
- **To Cache Mem:** As shown in Figure 3.5a, this separate module stores the actual data. The controller sends indices and write enables to this block.
- **To Main Memory:** As shown in Figure 3.5b, this interface handles block refills (512 bits) and write-through operations.



(a) Cache Memory (Data Store)

(b) Main Memory Interface

Figure 3.5: Memory Storage Modules.

3.2 Foundational Memory Management Formulas

The RTL design is guided by standard addressing formulas used to locate and validate data.

3.2.1 Cache Address Decomposition

The 32-bit Physical Address is decomposed as follows, based on the 8KB size and 64B blocks:

$$\text{Offset} = \log_2(64) = 6 \text{ bits} \tag{3.1}$$

$$\text{Index} = \log_2\left(\frac{8192}{64 \times 2}\right) = 6 \text{ bits} \tag{3.2}$$

$$\text{Tag} = 32 - 6 - 6 = 20 \text{ bits} \tag{3.3}$$

3.2.2 MMU Address Decomposition

For virtual-to-physical translation, the MMU splits the address based on the 4KB page size.

$$\text{Page\_Offset\_bits} = \log_2(4096) = 12 \text{ bits} \tag{3.4}$$

$$\text{VPN\_bits} = 32 - 12 = 20 \text{ bits} \tag{3.5}$$

3.2.3 Performance Metrics

The simulation environment allows us to verify the behavior that impacts Average Memory Access Time (AMAT).

$$\text{AMAT} = \text{Hit\_Time} + (\text{Miss\_Rate} \times \text{Miss\_Penalty}) \tag{3.6}$$

Our cycle-accurate simulation confirms a Hit Time of 1 clock cycle and a variable Miss Penalty simulated by the main memory model.

3.3 Implementation Highlights and Code Analysis

Several advanced implementation details were critical to the project's success.

3.3.1 Script-Driven Verification Environment

A unique feature of this project is the implementation of a file-based instruction reader in the testbench. Instead of hardcoding test vectors, the testbench uses Verilog file I/O system tasks (`$fopen`, `$fscanf`) to read a text file ('instructions.txt') containing memory operations.

Instruction Format:

```
R 0x1000      // Read from 0x1000
W 0x2000 A    // Write 0xA to 0x2000
```

This approach allows for the rapid creation of complex test scenarios without recompiling the Verilog code.

3.3.2 Coherence and Invalidation Logic

During the implementation of the Write-Through policy, we observed a potential data inconsistency. To resolve this, we implemented an **Invalidation Protocol**. On any Write Hit, the controller explicitly invalidates the matching cache line.

```
// Invalidate cache line on Write Hit
if (state == S_CHECK_HIT && is_hit && reg_is_write) begin
     if (way0_hit) valid_store[addr_index][0] <= 1'b0;
     if (way1_hit) valid_store[addr_index][1] <= 1'b0;
end
```

Listing 3.1: Invalidation Logic in Verilog

This forces the next read to the same address to result in a Cache Miss, triggering a fetch from main memory which retrieves the updated data, ensuring coherence.

3.4 Simulation Observations

During the verification phase, several critical behaviors were observed and corrected.

3.4.1 Observation 1: Reset Logic and Verilog Standards

Initial simulations showed that internal storage arrays (`tag_store`, `lru_store`) remained in an unknown state (`X`) after reset. Analysis revealed this was due to the use of Verilog-2001 syntax (`integer i=0` inside a `for` loop) with a simulator defaulting to Verilog-1995 standards. The code was refactored to declare loop variables at the module scope, resolving the initialization issue.

3.4.2 Observation 2: Hit/Miss Signal Timing

An important timing behavior was observed regarding the `hit_miss` output signal. The testbench initially checked this signal after the controller returned to the `IDLE` state. However, by that time, the cache had already refilled the missing line, causing the logic to report a "Hit" for what was actually a "Miss" event. We updated the verification logic to capture the `hit_miss` signal state *during* the request cycle, allowing for accurate hit/miss statistics.

Chapter 4

Future Work

4.1 Future Objectives

While the RTL design and functional verification are complete, the ultimate goal of this project is to proceed through the complete ASIC design flow towards tapeout. The future work is therefore focused on the physical implementation stages.

1. **Logic Synthesis:** The verified RTL will be synthesized using Synopsys Design Compiler to generate a gate-level netlist mapped to a standard cell library (e.g., 45nm or 28nm technology node). This step will provide concrete area, power, and timing estimates.

2. **Static Timing Analysis (STA):** Comprehensive timing analysis will be performed using Synopsys PrimeTime to ensure the design meets setup and hold time constraints across various process, voltage, and temperature (PVT) corners.

3. **Floorplanning and Placement:** We will define the physical footprint of the chip, placing memory macros (SRAMs) and standard cells to optimize wire length and congestion.

4. **Clock Tree Synthesis (CTS):** A robust clock distribution network will be inserted to minimize clock skew and insertion delay across the chip.

5. **Routing and Physical Verification:** The final interconnections will be routed, followed by Design Rule Checking (DRC) and Layout vs. Schematic (LVS) checks to ensure the physical layout matches the logical design and manufacturing constraints.

6. **GDSII Generation:** The final deliverable will be the GDSII stream file, ready for foundry fabrication (tapeout).

Bibliography

[1] S. R. Sarangi, *Computer Architecture and Organization*. McGraw-Hill Education, 2017.

[2] SHAKTI Processor Project, "SHAKTI Cache Controller Implementation in Bluespec," `https://gitlab.com/shaktiproject/uncore/caches`.

[3] Branch Education, "Cache Memory and RAM Explained," `https://youtu.be/7J7X7aZvMXQ?si=TxehuOpF31d4Nuxu`.

[4] Tech With Nikola, "But, what is Virtual Memory?", `https://youtu.be/A9WLYbEOp-I?si=BJN-8I-X92I3BC5-`.

[5] Geeks for Geeks, "Cache Memory in Computer Organization," `https://www.geeksforgeeks.org/cache-memory-in-computer-organization/`.

[6] Geeks for Geeks, "Virtually Indexed Physically Tagged (VIPT) Cache," `https://www.geeksforgeeks.org/computer-organization-architecture/virtually-indexed-physically-tagged-vipt-cache/`.