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

SHIV NADAR
INSTITUTION OF EMINENCE DEEMED TO BE
UNIVERSITY
DELHI NCR

SCHOOL OF
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 objectives achieved in this project are:

1. **RTL Implementation:** Development of synthesizable Verilog modules for the MMU and Cache Controller.

2. **FSM Control:** Implementation of a deterministic Finite State Machine to manage the cache life-cycle (Hit, Miss, Refill, Write-Through).

3. **Script-Driven Verification:** Creation of a flexible verification environment that reads memory access patterns from an external text file.

4. **Coherence Logic:** Implementation of cache line invalidation to maintain consistency between the cache and main memory during write operations.

1.3 Design Specifications

The system configuration is detailed in Table 1.1.

Table 1.1: 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 2

System Architecture and Design

2.1 Overall Architecture

The memory subsystem operates as a pipeline. The CPU (simulated) issues a Virtual Address (VA). The MMU translates this VA into a Physical Address (PA) using its internal TLB or by stalling the CPU to walk the page tables (simulated delay). The PA is then forwarded to the Cache Controller, which checks its internal SRAMs for a hit or fetches data from Main Memory.

2.2 Memory Management Unit (MMU)

The MMU is responsible for address translation and protection.

2.2.1 Translation Lookaside Buffer (TLB)

The core of the MMU is a fully associative TLB (`tlb_simple.v`). A fully associative structure was chosen to maximize the hit rate for the small number of entries (4 entries in the current implementation).

- **Lookup:** The incoming Virtual Page Number (VPN) is compared against all valid tags in parallel.
- **Refill:** On a miss, the `ptw_miss_detected` signal initiates a page table walk. The testbench simulates the retrieval of the Physical Frame Number (PFN), which is then written into the TLB using a Round-Robin replacement pointer.

2.3 Cache Controller Design

The Cache Controller manages the 8KB data store and the tag arrays. It is designed with modularity in mind, separating the control logic (FSM) from the memory arrays.

2.3.1 Finite State Machine (FSM)

A robust 7-state FSM governs the controller's operation:

- **S_IDLE:** Default state, waiting for `read_mem` or `write_mem` signals.
- **S_CHECK_HIT:** Performs tag comparison. If `is_hit` is high, data is returned immediately.
- **S_READ_MISS_FETCH/WAIT:** On a miss, these states assert `main_mem_read_req` and wait for the `main_mem_ready` handshake.
- **S_READ_MISS_REFILL:** Updates the cache SRAM with the new block, updates the Tag Store, and flips the LRU bit.
- **S_WRITE_THROUGH:** Handles write requests by forwarding them to main memory.

2.3.2 Cache Addressing

The 32-bit address is decomposed based on the cache geometry:

$$\text{Offset} = \log_2(64) = 6 \text{ bits} \tag{2.1}$$

$$\text{Index} = \log_2\left(\frac{8192}{64 \times 2}\right) = 6 \text{ bits} \tag{2.2}$$

$$\text{Tag} = 32 - 6 - 6 = 20 \text{ bits} \tag{2.3}$$

Chapter 3

Implementation Details

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: The parser supports a simple mnemonic-based format:

- `R <Address>`: Perform a Read at the hex address.
- `W <Address> <Data>`: Write the hex data to the hex address.

Example `instructions.txt`:

```
R 0x1000      // Read from 0x1000 (Expect Miss)
W 0x2000 A    // Write 0xA to 0x2000
R 0x2000      // Read back (Expect Hit with data 0xA)
R 0x4000      // Read new address (Expect Miss)
```

This approach allows for the rapid creation of complex test scenarios (e.g., thrashing a specific set to test LRU logic) without recompiling the Verilog code, significantly speeding up the verification process.

3.2 Modularity and Scalability

The design explicitly separates the **Control Plane** (FSM) from the **Data Plane** (SRAM arrays).

```verilog
// Interface to Cache Memory (Data Plane)
output reg[5:0]  cache_mem_index,
output reg [511:0] cache_mem_data_in,
output reg cache_mem_write_en,
input wire [511:0] cache_mem_data_out
```

Listing 3.1: Modular Interface Definition

By defining these distinct interfaces, the underlying memory implementation can be swapped (e.g., from register arrays to BRAMs on an FPGA) without modifying the complex FSM logic. This also prepares the architecture for a multi-level hierarchy, where the "Main Memory" interface can simply be connected to an L2 Cache Controller.

3.3 Coherence and Invalidation Logic

During the implementation of the Write-Through policy, we observed a potential data inconsistency. Since our design does not allocate a cache line on a write miss (No-Write-Allocate), and simply writes to main memory, a subsequent read might return stale data if the line was already present in the cache but not updated.

To resolve this, we implemented an **Invalidation Protocol**. On any Write Hit, the controller explicitly invalidates the matching cache line.

```verilog
// 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 3.2: 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, thus ensuring coherence.

Chapter 4

Simulation Observations and Analysis

4.1 Simulation Environment

The design was simulated using **Synopsys VCS** in a Linux environment. The waveform analysis was conducted using Verdi.

4.2 Key Factual Observations

Several critical behaviors were observed and corrected during the verification phase.

4.2.1 Observation 1: Reset Logic and Verilog Standards

Initial simulations showed that the 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, ensuring backward compatibility and successful initialization of all arrays to `0`.

4.2.2 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 combinatorial logic to report a "Hit" for what was actually a "Miss" event. **Resolution:** The verification logic was updated to capture the `hit_miss` signal state *during* the request cycle (`@(posedge clk)`). This allows the testbench to accurately distinguish between a request that finds data (Hit) and one that triggers a fetch (Miss).

4.2.3 Observation 3: Cycle-Accurate Memory Latency

To realistically model the "memory wall," the testbench implementation of Main Memory avoids simple blocking delays (`#30`). Instead, it implements a cycle-counter state machine.

- When a read request is received, the memory model enters a `WAIT` state for 3 clock cycles.
- During this time, the Cache Controller correctly asserts `ready_stall`, freezing the CPU pipeline.
- This confirms the controller's ability to handle variable-latency memory responses without data loss.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This project has successfully delivered a functional, synthesizable RTL implementation of a memory management subsystem. The design meets all functional requirements, demonstrating correct address translation, effective caching via the 2-way set-associative architecture, and robust handling of memory hazards via the FSM. The introduction of the script-driven verification environment and the modular design approach highlights a focus on scalability and verification efficiency, distinguishing this work from standard academic implementations.

5.2 Future Work

While the current system is functional, several enhancements are planned:

1. **Write-Back Policy:** Implementing a ”Dirty Bit” to support Write-Back, which will significantly reduce traffic to main memory by only writing modified data upon eviction.

2. **L2 Cache Integration:** Utilizing the modular interface to connect a Unified L2 Cache, creating a deeper memory hierarchy.

3. **FPGA Prototyping:** Synthesizing the design for a Xilinx Artix-7 FPGA to measure real-world resource utilization (LUTs/BRAMs) and timing performance.

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

[4] Arora, Narain D et al., "Electron and hole mobilities in silicon as a function of concentration and temperature," *IEEE Trans. Electron Devices*, vol. 29, no. 2, 1982.