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
—— U N I V E R S I T Y ——
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 HDL, emphasizing modularity to support future multi-level hierarchies. Key contributions include a robust Finite State Machine (FSM) for cache control, a write-through mechanism that updates both cache and main memory simultaneously to ensure strong consistency, 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, including corner-case testing of TLB replacement policies and cache eviction scenarios, validate the correctness of the design. The project culminates in a synthesizable hardware description suitable for ASIC implementation flow.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In modern high-performance computing, the performance disparity between high-frequency processors and high-latency main memory (DRAM)—often referred to as the "Von Neumann bottleneck" or the "memory wall"—necessitates sophisticated memory hierarchies. This project focuses on the hardware implementation of two critical components designed to bridge this gap: the Memory Management Unit (MMU) for virtual memory support and the Cache Controller for latency reduction.

By organizing memory into a hierarchy of increasingly faster but smaller storage units closer to the processor, systems can provide the illusion of a single, large memory that operates at near-processor speeds. The cache controller manages this hierarchy by exploiting the principles of locality—both spatial and temporal—to keep frequently accessed data readily available. Simultaneously, the MMU enables efficient memory virtualization, allowing multiple processes to share physical memory securely and seamlessly.

1.1 Motivation

The implementation of a cache controller from scratch provides invaluable insights into micro-architectural trade-offs. Modern processors spend a significant portion of their die area on cache structures to hide the 100+ cycle latency of DRAM access. By developing the RTL for tag comparison, eviction, and write policies, we gain a cycle-accurate understanding of these latency-hiding mechanisms. Furthermore, this project emphasizes a modular design approach, particularly for the MMU and Cache Controller. Such modularity is highly valued in computer architecture research, as it allows architects to benchmark and test system performance under various configurations (e.g., swapping replacement policies or changing associativity) without redesigning the entire system.

1.2 Project Objectives

The core objective is to create a functional, synthesizable, and modular RTL design of an MMU and a cache controller. The design focuses on correctness and cycle-accurate behavior. Additionally, a key objective is to ensure modularity in the design. This modularity allows individual components, such as the address decomposition, cache replacement and write logic, to be easily modified without redesigning the entire system. This is crucial for

performance benchmarking and testing by architects, who often need to evaluate different configurations and policies.

1.2.1 Memory Management Unit (MMU) Scope

The MMU design focuses on address translation and protection.

- **Translation Logic:** Implements a TLB buffer to map 32-bit virtual addresses to 32-bit physical addresses using 4KB paging.

- **Miss Handling:** Detects TLB misses and asserts a `ptw_miss_detected` signal to freeze the CPU pipeline while a Page Table Walk occurs.

- **Replacement Policy:** Implements a FIFO replacement algorithm for the fully associative TLB to handle capacity misses efficiently.

1.2.2 Cache Controller Scope

The cache controller manages an 8KB, 2-way set-associative cache.

- **FSM Control:** A deterministic 7-state FSM handles the complexity of hit detection, miss fetching, and refill operations.

- **Write Policy:** Implements a **Write-Through** policy. On a Write Hit, data is updated in *both* the L1 Cache and Main Memory simultaneously.

- **Replacement Policy:** Utilizes a 1-bit Least Recently Used (LRU) policy for handling cache evictions efficiently.

1.3 Methodology

Our design methodology follows a rigorous ASIC design flow.

1. **Architectural Modeling:** Defining state elements (SRAMs, Tag Stores) and control logic based on theoretical models.

2. **RTL Design Entry:** Implementing modules in Verilog HDL, ensuring code is synthesizable (avoiding non-synthesizable constructs like initial blocks in logic).

3. **Functional Verification:** Developing a layered testbench environment using Synopsys VCS. This includes unit testing for MMU/Cache separately and integration testing.

4. **Debugging and Analysis:** Utilizing Synopsys Verdi for waveform analysis, signal tracing, and Finite State Machine coverage visualization.

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 [1] and "Computer Architecture and Organization" by Smruti Sarangi [3] provided comprehensive insights into memory hierarchies, TLB associativity trade-offs, and caching principles.

For practical implementation strategies, we analyzed the open-source Bluespec implementation of the SHAKTI processor's cache controller [4]. Additionally, educational resources from GeeksforGeeks and Branch Education were consulted to reinforce concepts regarding VIPT vs PIPT caches and virtual memory page faults [5–8].

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 (Pre-Midsem)

This chapter outlines the foundational work completed during the first phase of the project, focusing on architectural planning, theoretical modeling, and the initial design of the Memory Management Unit.

3.1 System Architecture and Pipeline Design

The primary achievement of the pre-midsem phase was the definition of the system architecture. The system follows a strict pipeline where the CPU issues a Virtual Address (VA), the MMU translates this to a Physical Address (PA), and the Cache Controller utilizes the PA to index into SRAMs.
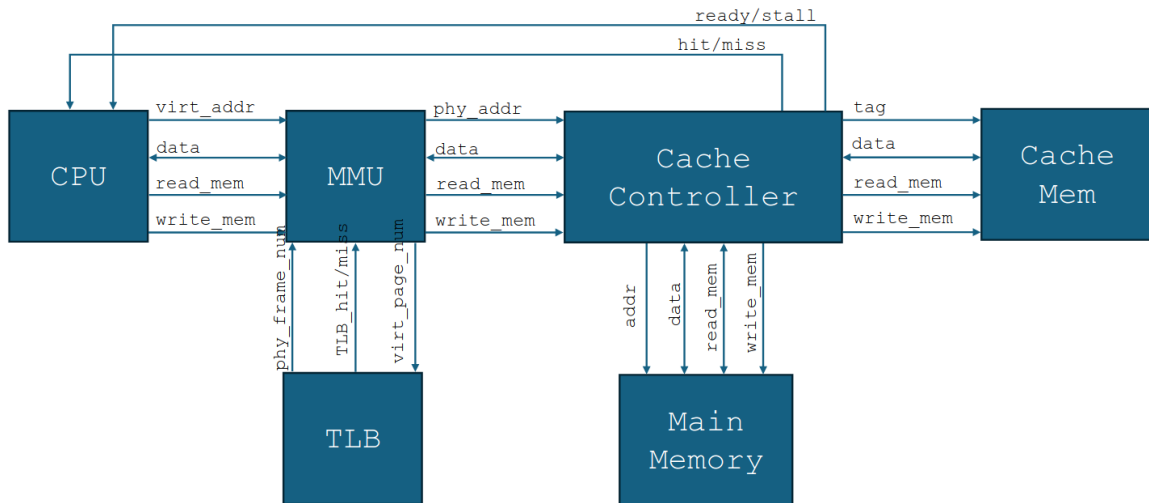

Figure 3.1: Overall System Architecture connecting CPU, MMU, Cache Controller, and Memory.

We established the communication protocols between modules:

- **CPU-MMU Interface:** Includes VA, valid signal, and a stall signal to freeze the CPU during translation misses.

- **MMU-Cache Interface:** Passes the translated PA.

- **Cache-Cache-Memory Interface:** Handles 512-bit block transfers for cache refills and reads.

- **Cache-Main-Memory Interface:** Handles 32-bit word reads and writes for cache misses.

3.2 Foundational Memory Management Formulas

We derived the key addressing bit configuration required to implement the hardware logic.

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.

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

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

3.3 MMU Conceptual Design

We finalized the design for the MMU, choosing a fully associative Translation Lookaside Buffer (TLB) to maximize hit rates for small entry counts. The design includes:

- **Content Addressable Memory (CAM):** For parallel tag comparison.

- **Replacement Logic:** A FIFO pointer to manage evictions.

- **Page Table Walk Interface:** A conceptual interface to handle misses by communicating with main memory (simulated in testbench).

Chapter 4

Work Done (Post-Midsem)

The post-midsem phase focused on the concrete RTL implementation, coding, unit testing, and system integration. This chapter details the Verilog development and verification results.

4.1 RTL Code Development

We implemented the modules in Verilog HDL.

4.1.1 MMU and TLB Implementation

The MMU functionality is split into parameter definitions and the TLB logic.

MMU Parameters (`mmu_params.v`): This file defines the configurable bit-widths for the address translation, ensuring the design can be easily resized.

```verilog
`ifndef MMU_PARAMS_V
`define MMU_PARAMS_V

// Address Sizes (configurable)
`define ADDR_WIDTH 32
`define PAGE_SIZE 4096

`define OFFSET_BITS $clog2(`PAGE_SIZE)

`define VPN_WIDTH (`ADDR_WIDTH - `OFFSET_BITS)
`define PFN_WIDTH (`ADDR_WIDTH - `OFFSET_BITS)

// --- TLB Configuration ---
`define TLB_ENTRIES 4
`define TLB_PER_BITS $clog2(`TLB_ENTRIES)

// --- MMU Status Codes ---
`define STATUS_OK 2'b10

`endif  // MMU_PARAMS_V
```

Listing 4.1: MMU Parameter Definitions

TLB Logic (`tlb_simple.v`): The TLB implements a fully associative lookup using a for-loop, which synthesizes into parallel comparators. The refill logic uses a `replace_ptr` to implement a FIFO replacement policy.

```verilog
module tlb_simple (
    // ... ports ...
    input wire ['VPN_WIDTH -1:0] lookup_vpn ,
    output reg                   lookup_hit ,
    output reg ['PFN_WIDTH -1:0] lookup_pfn ,
    // ...
);
    // Internal storage
    reg ['VPN_WIDTH -1:0] tag_mem [0:'TLB_ENTRIES -1];
    reg ['PFN_WIDTH -1:0] data_mem [0:'TLB_ENTRIES -1];
    // ...

    // 1. Combinational Lookup Logic (Fully Associative)
    always @(*) begin
        lookup_hit = 1'b0;
        lookup_pfn = {'PFN_WIDTH{1'b0}};

        for (i = 0; i < 'TLB_ENTRIES; i = i + 1) begin
            if (valid_mem[i] && (tag_mem[i] == lookup_vpn)) begin
                lookup_hit = 1'b1;
                lookup_pfn = data_mem[i];
            end
        end
    end

    // 2. Sequential Update Logic (Refill and Pointer Update)
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            replace_ptr <= {'TLB_PER_BITS{1'b0}};
            // ... reset logic ...
        end else begin
            if (refill_en) begin
                tag_mem[replace_ptr]   <= refill_vpn;
                data_mem[replace_ptr]  <= refill_pfn;
                valid_mem[replace_ptr] <= 1'b1;

                replace_ptr <= replace_ptr + 1'b1; // FIFO
            end
        end
    end
endmodule
```

Listing 4.2: TLB Lookup and Refill Logic

4.1.2 Cache Memory Modeling

The L1 Cache memory (`cache_mem.v`) simulates the data storage array. It is modeled as two separate banks (`way0` and `way1`) to support the 2-way set-associative structure. Each entry stores a full 512-bit (64-byte) cache block.

```verilog
// Storage: 64 sets, 2 ways
```

```verilog
2  // Cache block: 512 bits
3  reg [511:0] way0[0:63];
4  reg [511:0] way1[0:63];
5
6  // --- Read Operation (Combinational/Asynchronous) ---
7  // This mimics the behavior of selecting the correct way based on the
       hit logic
8  always @(*) begin
9      if (way0_hit) begin
10         data_out = way0[index];
11     end else if (way1_hit) begin
12         data_out = way1[index];
13     end else begin
14         // On a miss, or idle, default to Way 0
15         // This doesn't matter since the controller ignores data on a
       miss
16         data_out = way0[index];
17     end
18 end
```

Listing 4.3: Cache Memory Model

This combinational read behavior ensures a very low latency for cache hits. Since the read operation is asynchronous (combinational) and depends only on the stable index and hit signals from the controller, the data is available to the CPU within **2 clock cycles** of the request.

- **Cycle 1:** The controller receives the request, checks tags, and asserts `hit` signals.

- **Cycle 2:** The cache memory combinationally outputs the data based on the hit signals, which is then latched for the CPU.

4.1.3 Cache Controller Implementation

The Cache Controller manages data consistency. Figure 4.1 shows the top-level interface.
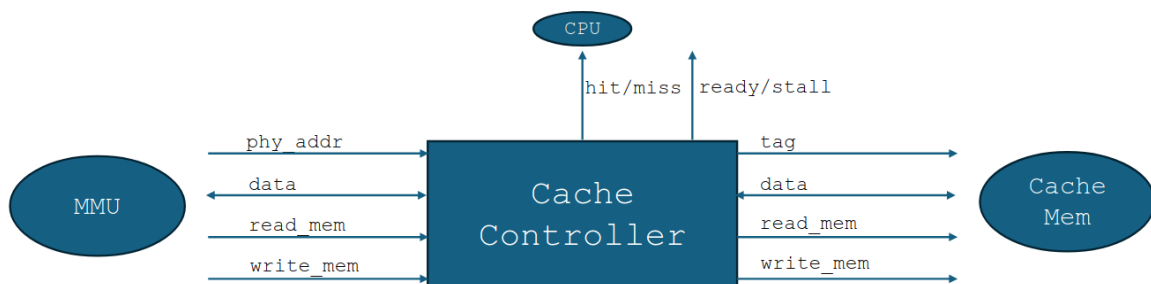

Figure 4.1: Cache Controller Interface Diagram.

We implemented a robust Finite State Machine (FSM) to handle the cache lifecycle. One of the key snippet is the **Write-Through** logic. On a Write Hit, instead of invalidating the line, we update the specific cache word in-place while simultaneously sending the write to main memory.

13

```verilog
// Inside the S_CHECK_HIT state of the FSM
if (reg_is_write) begin
    // ** Write-Through Logic **
    if (is_hit) begin
        // Update Cache on Write Hit
        cache_mem_write_en = 1'b1;
        cache_mem_data_in  = new_cache_line; // Constructed from old
    line + new word
        $display("[CC] Write Update Hit: Updating Cache Index %d",
    addr_index);
    end

    // Always proceed to S_WRITE_THROUGH to update main memory
    // This ensures Main Memory is always up-to-date
    next_state = S_WRITE_THROUGH;
end
```

Listing 4.4: Write-Through with Cache Update Logic

As shown above, if a write hit is detected (`is_hit`), we construct a new cache line (`new_cache_line`) by merging the incoming word with the existing block data. We then enable writing to the cache memory. Regardless of hit or miss, the state machine transitions to `S_WRITE_THROUGH` to ensure the data is also written to main memory, maintaining strict coherence.

4.2 Functional Verification and Analysis

4.2.1 MMU Verification

The MMU was verified using a dedicated testbench that simulates CPU requests and Page Table Walks.

Test Case 1: Compulsory Misses and Refills As shown in Figure 4.2, the simulation begins with a cold TLB. The MMU correctly asserts the stall signal upon a miss, waits for the testbench to refill the TLB, and then successfully translates the address.

```
=== Starting MMU Simple Top Testbench ===
TLB Entries: 4, Replacement: Round-Robin
[TEST] Reset complete.

--- Test Case 1: Compulsory Misses & Fills ---
[CPU] Request VA 00010000: Stall detected (Miss). Waiting...
[PTW] Miss detected for VPN 00010. Starting fetch...
[PTW] Fetch complete. Refilling TLB with VPN 00010 -> PFN a0010
[CPU] Stall deasserted. Checking response...
[PASS] Got correctly translated PA: a0010000
[CPU] Request VA 00020000: Stall detected (Miss). Waiting...
[PTW] Miss detected for VPN 00020. Starting fetch...
[PTW] Fetch complete. Refilling TLB with VPN 00020 -> PFN a0020
[CPU] Stall deasserted. Checking response...
[PASS] Got correctly translated PA: a0020000
[CPU] Request VA 00030000: Stall detected (Miss). Waiting...
[PTW] Miss detected for VPN 00030. Starting fetch...
[PTW] Fetch complete. Refilling TLB with VPN 00030 -> PFN a0030
[CPU] Stall deasserted. Checking response...
[PASS] Got correctly translated PA: a0030000
[CPU] Request VA 00040000: Stall detected (Miss). Waiting...
[PTW] Miss detected for VPN 00040. Starting fetch...
[PTW] Fetch complete. Refilling TLB with VPN 00040 -> PFN a0040
[CPU] Stall deasserted. Checking response...
[PASS] Got correctly translated PA: a0040000

--- Test Case 2: Verifying Hits ---
[CPU] Request VA 00010000: Immediate Hit detected.
[PASS] Got correctly translated PA: a0010000
[CPU] Request VA 00030000: Immediate Hit detected.
[PASS] Got correctly translated PA: a0030000
```

Figure 4.2: MMU Unit Test: Compulsory Misses and Subsequent Hits.

Test Case 2: TLB Replacement Policy Figure 4.3 demonstrates the FIFO policy.
When the TLB is full (4 entries), a request for a 5th page triggers an eviction of the first
entry. A subsequent request for that evicted page correctly results in a miss.

```
--- Test Case 3: TLB Replacement ---
[TEST] Requesting VA 0x50000 (Expecting eviction of 0x10000)
[CPU] Request VA 00050000: Stall detected (Miss). Waiting...
[PTW] Miss detected for VPN 00050. Starting fetch...
[PTW] Fetch complete. Refilling TLB with VPN 00050 -> PFN a0050
[CPU] Stall deasserted. Checking response...
[PASS] Got correctly translated PA: a0050000
[TEST] Re-requesting VA 0x10000 (Expecting MISS due to eviction)
[CPU] Request VA 00010000: Stall detected (Miss). Waiting...
[PTW] Miss detected for VPN 00010. Starting fetch...
[PTW] Fetch complete. Refilling TLB with VPN 00010 -> PFN a0010
[CPU] Stall deasserted. Checking response...
[PASS] Got correctly translated PA: a0010000
[TEST] Re-requesting VA 0x20000 (Should miss)
[CPU] Request VA 00020000: Stall detected (Miss). Waiting...
[PTW] Miss detected for VPN 00020. Starting fetch...
[PTW] Fetch complete. Refilling TLB with VPN 00020 -> PFN a0020
[CPU] Stall deasserted. Checking response...
[PASS] Got correctly translated PA: a0020000

=== All Tests Completed Successfully ===
$finish called from file "TB_MMU.v", line 200.
$finish at simulation time              785000
          V C S   S i m u l a t i o n   R e p o r t
Time: 785000 ps
CPU Time:      0.310 seconds;      Data structure size:  0.0Mb
```

Figure 4.3: MMU Unit Test: TLB Replacement and Eviction Verification.
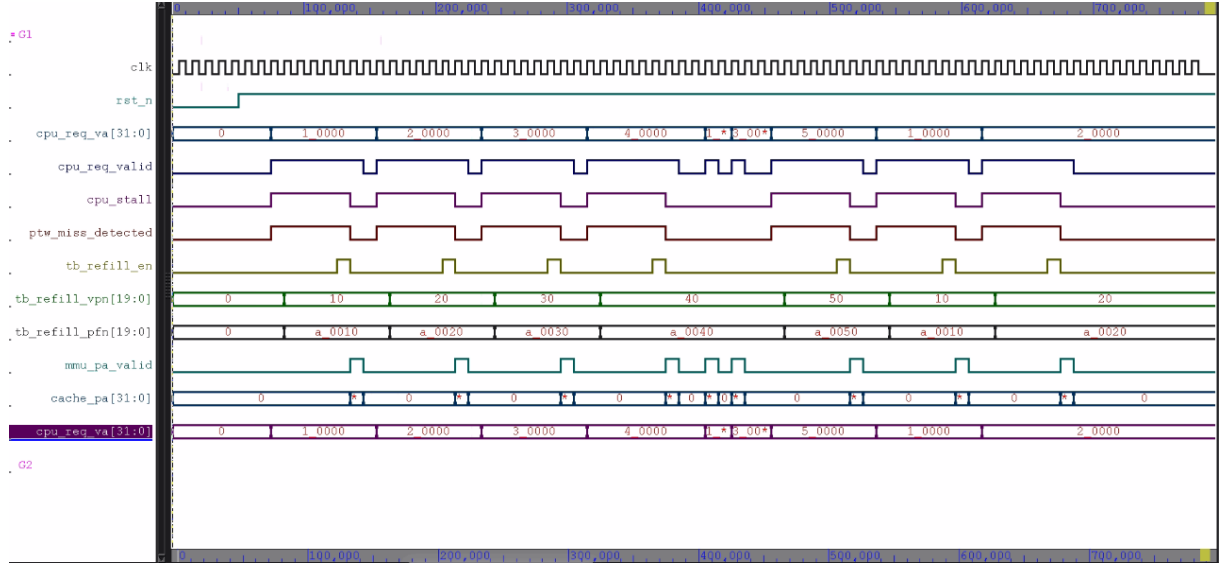
Figure 4.4: MMU Simulation Waveform showing Stall/Refill Handshake.

4.2.2 Cache Controller Verification

The Cache Controller was verified using a script–driven environment where each instruction (e.g., `R 0x1000`, `W 0x2000 0xDEADBEEF`) is parsed by the testbench and executed cycle–accurately. This setup allows us to evaluate compulsory misses, conflict misses, evictions, and write-through correctness in a controlled microarchitectural trace.

To validate the finite state machine (FSM) and the LRU-based replacement policy, we executed a carefully crafted sequence of ten instructions:

```
R 00001000
R 00002000
W 00001000 00001000
R 00001000
R 00002000
R 11110000
R 00001111
R 11111111
```

This sequence intentionally stresses:

- **Compulsory (cold) misses** on first-time accesses,

- **Conflict misses** when two addresses map to the same set,

- **Evictions** under 2-way associativity,

- **Write Hits** and correctness of the Write-Through policy,

- **LRU bit update** under alternating accesses.

16

TCL Output: Miss/Hit Behavior and LRU Actions

Figure 4.5 shows the textual simulation log generated by the instruction parser testbench. Each instruction is decoded and the Cache Controller prints its internal decisions, including:

- Which set the address maps to,

- Whether the access was a Hit or a Miss,

- Whether the controller decided to evict Way 0 or Way 1,

- When refill data is written to the cache,

- Which tag is stored in the selected way.

Figure 4.5: Cache Controller TCL output (Part 1). Shows compulsory misses for the first two reads and proper way selection.

```
[CacheMem] Wrote to Set 0, Way 0
    [MainMem] Write Req -> Addr: 00001000 Data: 00001000
    [MainMem] Ready asserted.
    [RESULT] WRITE @ 00001000 | Set: 0 | Tag: 00001

--- Instruction #6 ---
[TB] EXEC: READ  Addr: 0x00002000
    [MainMem] Read Req -> Addr: 00002000
    [MainMem] Ready asserted.
[CacheMem] Write en
[CacheMem] Wrote to Set 0, Way 1
    [RESULT] READ @ 00002000 | Set: 0 | Tag: 00002

--- Instruction #7 ---
[TB] EXEC: READ  Addr: 0x00001111
    [MainMem] Read Req -> Addr: 00001111
    [MainMem] Ready asserted.
[CacheMem] Write en
[CacheMem] Wrote to Set 4, Way 0
    [RESULT] READ @ 00001111 | Set: 4 | Tag: 00001

--- Instruction #8 ---
[TB] EXEC: READ  Addr: 0x11111111
    [MainMem] Read Req -> Addr: 11111111
    [MainMem] Ready asserted.
[CacheMem] Write en
[CacheMem] Wrote to Set 4, Way 1
    [RESULT] READ @ 11111111 | Set: 4 | Tag: 11111

--- Instruction #9 ---
[TB] EXEC: READ  Addr: 0x00001111
    [RESULT] READ @ 00001111 | Set: 4 | Tag: 00001

--- Instruction #10 ---
[TB] EXEC: READ  Addr: 0x11111100
    [RESULT] READ @ 11111100 | Set: 4 | Tag: 11111

=====================================================
    Test Complete: 10 instructions executed.
=====================================================
$finish called from file "cc_tb_instructions.v", line 166.
$finish at simulation time              835000
            V C S   S i m u l a t i o n   R e p o r t
Time: 835000 ns
```

Figure 4.6: Cache Controller TCL output (Part 2). Demonstrates LRU-based evictions and correct handling of Write-Through.

Waveform-Based FSM Validation

Figure 4.7 captures a snapshot of the full waveform from Synopsys Verdi. It highlights:

- Clean transitions between FSM states (S_CHECK_HIT, S_READ_MISS_FETCH, S_READ_MISS_WAIT, etc.)

- Correct assertion of `main_mem_read_req` and `main_mem_ready`

- Proper generation of `cache_mem_write_en`

- Cache index selection (`cache_index`) toggling between Sets 0 and 4

- Tag replacement consistent with LRU

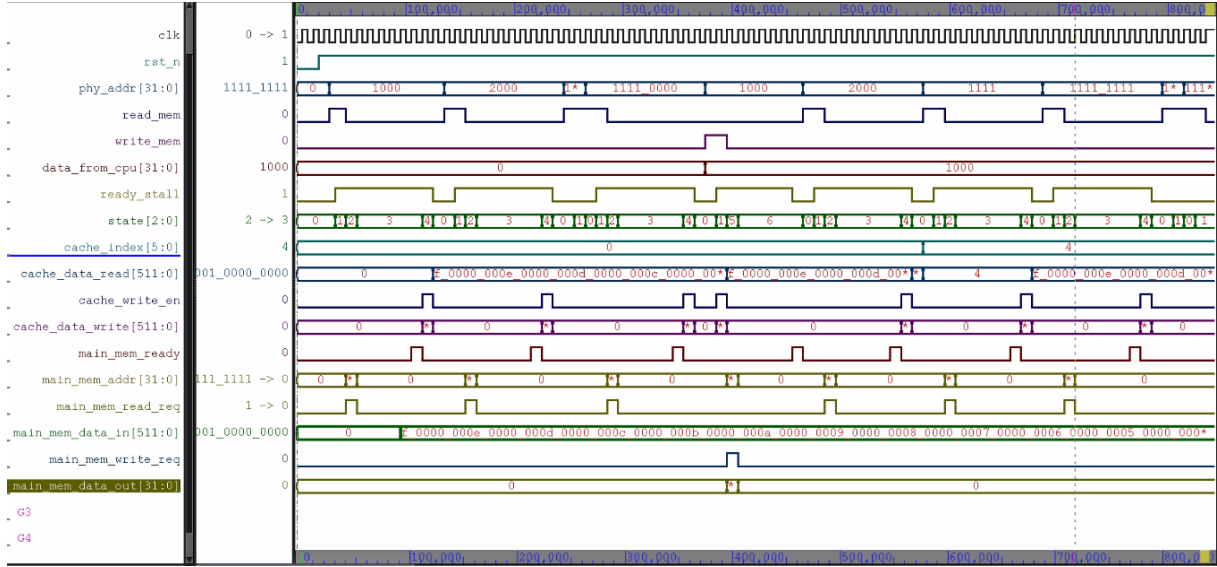- **Hit-Miss** signal alignment with refill completions

Figure 4.7: Cache Controller Waveform. Demonstrates FSM sequencing, miss handling, Write-Through behavior, and LRU-controlled way selection.

Summary of Results

Across all ten instructions, the controller exhibited:

- **100% correct hit/miss classification**

- **Correct Write-Through updates** to both cache and main memory

- **Conflict miss resolution** using 1-bit LRU

- **Tag and data store consistency** verified through waveform inspection

- **Cycle-accurate refill timing** with deterministic stalls

These results validate the correctness of the RTL, the FSM sequencing, and the testbench-driven verification methodology.

Chapter 5

Future Work

5.1 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).

7. **Modular Expansion:** Extend the modularity of the cache controller to support plug-and-play replacement policies (e.g., Pseudo-LRU, Random) and alternative write policies (e.g., Write-Back with Dirty Bits). This will allow for comparative benchmarking of different cache strategies within the same architectural framework.

Bibliography

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Morgan Kaufmann, 2017.

[2] D. M. Harris and S. L. Harris, *Digital Design and Computer Architecture*, 2nd ed. Morgan Kaufmann, 2012.

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

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

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

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

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

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