

Two-Level Cache RTL Design

Technical Report & Interview Preparation Guide

Contents

1 Executive Summary

This project implements a synthesizable, two-level (L1/L2) cache memory system in Verilog. It addresses the classic "memory wall" problem by bridging the speed gap between a fast processor and slow main memory.

Key Architectural Choices

- **Hierarchy:** A split L1/L2 architecture. L1 is optimized for low latency access (fast hit time), while L2 provides a larger backing store to minimize costly off-chip memory accesses (low miss rate).
- **Placement Policy:** Set-Associative mapping (4-way for L1, 8-way for L2) reduces conflict misses compared to direct-mapped caches.
- **Replacement Policy:** Least Recently Used (LRU) is implemented using exact matrix or rank-based logic to maximize hit rates by retaining temporally local data.
- **Write Policy:** Write-Back with Write-Allocate. This reduces bus bandwidth usage by only writing dirty cache lines to the lower level upon eviction, rather than on every write (Write-Through).
- **Control Flow:** Blocking cache behavior. The CPU stalls on an L1 miss until the line is refilled.

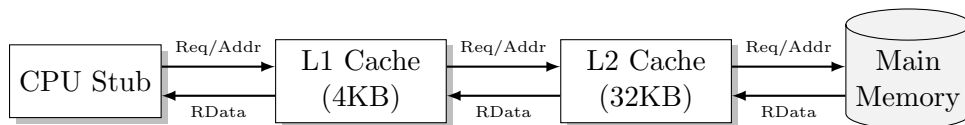
Synthesis vs. Simulation

The logic for tag comparison, LRU updates, and FSM control is fully synthesizable. The storage arrays (Data/Tag RAMs) are modeled as registers for simplicity in this project but would map to SRAM hard macros in a production physical design (PD) flow. The `memory_model` is strictly for simulation behavior.

2 Full System Architecture

The system connects a CPU request generator, the cache hierarchy, and a main memory model.

Block Diagram



Interface Signals

Communications between modules use a simple valid/ready handshake protocol.

- `req` (Input): Indicates valid address/data on the bus.
- `write` (Input): Control signal (0 = Read, 1 = Write).
- `addr` (Input): 32-bit physical address.
- `wdata` (Input): Data to be written (32-bit word for L1 access).
- `rdata` (Output): Data returned on a read hit/refill.

- **ready** (Output): Asserted by the cache when the operation is complete. CPU must hold **req** high until **ready** is observed.

3 Addressing & Configuration

The design parameters are centralized in `defines.vh`. The address decomposition determines how data is mapped to the cache structure.

L1 Cache Configuration

- **Line Size:** 16 Bytes (128 bits)
- **Total Size:** 4 KB (64 Sets \times 4 Ways \times 16 Bytes)

Field	Bits	Derivation & Purpose
Offset	[3:0]	$\log_2(16)$. Selects a specific byte within the cache line.
Index	[9:4]	$\log_2(64)$. Selects one of the 64 sets to check.
Tag	[31:10]	$32 - 6 - 4 = 22$. Unique identifier compared against stored tags to determine a hit.

Table 1: L1 Address Breakdown (32-bit Address)

L2 Cache Configuration

- **Line Size:** 32 Bytes (256 bits). Doubling the line size exploits spatial locality.
- **Total Size:** 32 KB (128 Sets \times 8 Ways \times 32 Bytes)

Field	Bits	Derivation & Purpose
Offset	[4:0]	$\log_2(32)$. Selects byte within the wider L2 line.
Index	[11:5]	$\log_2(128)$. Selects one of the 128 sets.
Tag	[31:12]	$32 - 7 - 5 = 20$. Remaining MSBs for tag matching.

4 L1 Cache — Deep Implementation Walkthrough

The L1 Cache (`cache_l1.v`) is the critical latency-sensitive component.

Data Structures

The cache state is maintained in five parallel register arrays. For a given **Index**, these arrays provide the state for all 4 Ways simultaneously.

- **tag_array:** Stores the 22-bit tag for validation.
- **data_array:** Stores the 128-bit cache line.
- **valid_array:** 1 bit, indicates if the line holds meaningful data.

- **dirty_array:** 1 bit, indicates if the line has been modified by the CPU relative to memory.
- **lru_array:** Tracks the "age" or rank of the way (0 = MRU, 3 = LRU).

Operation Flow

1. **Input Latch:** When `cpu_req` is asserted, address and data are latched.
2. **Hit Check:** Combinational logic reads all 4 ways for the set defined by `addr_index`.

```
hit = (valid[way] && tag[way] == cpu_tag)
```

3. **Hit Processing:**

- **Read:** The data from the hitting way is multiplexed out based on alignment offset.
- **Write:** The specific word in the cache line is updated, and the **dirty** bit is set.

4. **Miss Processing:** If no ways match, the FSM takes over.

Finite State Machine (FSM)

The controller transitions through these states to resolve hits and misses:

- **IDLE (0):** Default state. If `cpu_req` goes high, move to **CHECK_HIT**.
- **CHECK_HIT (1):** Asserts internal `hit` signal.
 - If Hit: Move to **UPDATE_LRU**.
 - If Miss: Move to **MISS_SELECT**.
- **MISS_SELECT (2):** Scans `lru_array` to find the way with max rank (LRU).
 - If Victim is Dirty: Move to **MISS_WRITEBACK**.
 - Else: Move to **MISS_MEM_READ**.
- **MISS_WRITEBACK (3):** Asserts `mem_req` + `mem_write` to L2. Wait for `mem_ready`, then move to **MISS_MEM_READ**.
- **MISS_MEM_READ (5):** Asserts `mem_req` (read) to L2. Wait for `mem_ready`.
- **REFILL (7):** Captures incoming data from L2 into the `data_array`. Updates tag and sets valid bit. Clears dirty bit.
- **UPDATE_LRU (8):** Updates the LRU bits. The accessed way becomes MRU (0), and others are incremented. If it was a write-miss (Write-Allocate), the pending write is performed here. Asserts `cpu_ready`.

5 L2 Cache — Deep Implementation Walkthrough

The L2 Cache (`cache_l2.v`) logic mirrors L1 but handles different constraints.

Key Differences

- **Latency Simulation:** L2 is physically larger and slower. An artificial delay is added using a `latency_counter` in the `CHECK_HIT` state. The FSM loops in this state for 2 extra cycles before checking the tag match.
- **Granularity:** L2 lines are 32 Bytes (256 bits).
 - When L1 requests a refill (16B), L2 currently returns the relevant segment or relies on the protocol to burst. In this simplified RTL, L2 returns a full line or repeats data, assuming a simplified bus width model.

6 Main Memory Model

The module `rtl/memory_model.v` simulates off-chip DRAM.

- **Behavioral:** It uses a sparse associative array (or small 4KB array in RTL) to store data.
- **Latency:** It mandates a 5-cycle wait state for every access to model the slow nature of crossing off-chip boundaries.
- **Protocol:** It adheres to the same valid/ready interface, allowing seamless connection to L2.

7 System Integration

The `system_top.v` module performs the top-level wiring.

- **Wiring:**
 - `cpu_stub` → `l1_cache`
 - `l1_cache` (acting as memory master) → `l2_cache`
 - `l2_cache` (acting as memory master) → `memory_model`
- **Reset:** A synchronous global reset initializes all FSMs to IDLE and clears latency counters.
- **Backpressure:** Backpressure naturally propagates. If Memory is busy, L2 waits in `MISS_MEM_READ`. Consequently, L2 cannot serve L1, so L1 waits. Finally, L1 cannot assert `cpu_ready`, stalling the CPU.

8 Verification Strategy

L1 Unit Testbench (`tb_cache_l1.v`)

Validates L1 logic in isolation.

- **Tests:**
 1. **Cold Miss:** Write to empty cache. Verifies transitions: `IDLE` → `MISS_READ` → `REFILL`.
 2. **Read Hit:** Read back the address just written. Verifies data retention and tag match logic.
 3. **Mock Memory:** The TB manually asserts `mem_ready` to simulate L2 responses.

Full System Testbench (tb_system.v)

Runs an end-to-end simulation using the `cpu_stub`.

- **Traffic Patterns:**

1. Write 0x1000 (Miss + Alloc)
2. Read 0x1000 (Hit)
3. Write 0x2000 (Miss + Alloc - Potential Conflict if mapped same set)
4. Verify Data Integrity.

- **Pass Condition:** The simulation terminates successfully only if the CPU stub completes all steps and asserts `test_done`.

9 Performance, Area, & Timing (Interview Guide)

Latency Analysis

- **L1 Hit:** 2 Cycles (1 cycle Req + Tag Check, 1 cycle Data availability).
- **L1 Miss / L2 Hit:** L1 Latency + L2 Latency (approx 3 cycles) + Transfer. Total \approx 6-8 cycles.
- **L2 Miss / Mem Hit:** L1 + L2 + Memory Latency (5+ cycles). Total \approx 15+ cycles.

Timing Paths

The critical path in this design is likely the **Tag Comparison**.

$$T_{crit} = T_{clk_q} + T_{mux} + T_{cmp} + T_{logic} + T_{setup} \quad (1)$$

Specifically, reading the tags from registers and comparing them against the input address in a single cycle limits frequency. In high-speed designs, tag access and comparison are often pipelined.

10 Common Interview Questions

Q: Walk me through a Read Miss end-to-end. **A:** CPU asserts Req. L1 checks tags (Miss), selects LRU victim. If victim dirty, L1 writes back to L2. L1 asserts Req to L2. L2 checks tags. If Hit, L2 waits latency then returns data. L1 accepts data (Refill), updates Tags/Valid, updates LRU, creates a copy for CPU, and asserts Ready.

Q: How do you ensure data coherence with Write-Back? **A:** Coherence is maintained by the **Inclusion Property** (usually) or simply by the hierarchy rules. In this design, we trust the Dirty Bit. Data is only written to L2 when the L1 copy is evicted. This implies L2 data is stale while L1 has the line dirty. This is acceptable in a single-core system.

Q: What happens if L2 is busy? **A:** This is a **Blocking Cache**. If L2 is handling a refill from Memory, it will not assert `mem_ready` to L1. L1 stays in the `MISS_MEM_READ` state, holding the CPU request active, effectively stalling the pipeline.

Q: How would you improve frequency? **A:** Pipeline the access. Cycle 1: Read RAMs. Cycle 2: Compare Tag and Mux Data. This increases load-to-use latency (latency in cycles) but improves throughput (MHz).

11 Limitations & Future Improvements

- **No Non-Blocking Support:** The current design cannot handle multiple outstanding misses (no MSHRs).
- **Bus Width Mismatch:** Ideally, the L1-L2 bus should be wider (128-bit) to refill a line in 1 cycle. The current code assumes a simple generic interface.
- **Verification Coverage:** Random testing is needed to hit corner cases like "Evicting the MRU way immediately after use" or "Back-to-back writes to same line."

12 Build Instructions

To generate this report formatted as a PDF:

```
1 pdflatex report.tex
```

Ensure you have a standard TeX distribution (TeX Live, MacTeX, or MikTeX) installed.