**CACHE CONTROLLER PROJECT DOCUMENTATION**

**Names**: Ghincul Dan, Georgescu Catalin  
**University**: Universitatea Politehnica Timișoara

**Objective**

Our objective was to design, implement, and test a fully functional **4-way set-associative cache controller** using Verilog. This controller plays a critical role in improving processor performance by reducing the number of direct memory accesses. Our design follows a write-back, write-allocate policy and employs a custom-built Least Recently Used (LRU) replacement strategy. The controller was integrated into a system where it handles both read and write operations from the CPU and interacts with a simplified memory module.

We were tasked with building this controller around a **Finite State Machine (FSM)** that orchestrates the transitions between states such as IDLE, CHECK\_HIT, EVICT, ALLOCATE, and SEND\_TO\_CPU. Alongside the hardware modules, we created an extensive **testbench** to validate correct behavior under various scenarios, including cache hits, misses, evictions, and data replacements.

Our implementation is written in **Verilog**, and simulation and debugging were performed in **ModelSim**.

**Initial Architecture**

We began by defining the internal structure of each cache line. Since our block size is 64 bytes (or 512 bits), and we needed to store additional metadata like valid, dirty, LRU, and tag fields, we defined each cache block to be 537 bits wide. The breakdown of this format is as follows:

valid[536], dirty[535], LRU[534:533], tag[532:512], block[511:0]

This allowed us to store all necessary information for each of the 4 blocks in every set. With 128 sets, we created a 2D array cache[127:0][3:0], and accessed each line based on the index extracted from the CPU address.

In parallel, we started sketching the FSM logic. Our initial plan was to create a minimal set of states that could cover every cache operation. We hand-drew a diagram with transitions and key conditions. This visual helped us keep the logic tight while preparing to implement the behavioral control.

A diagram of a triangle

AI-generated content may be incorrect.

**Final Architecture**

As the implementation evolved, we modularized the design to keep the logic manageable and scalable. The final structure included the following modules:

* cache\_controller.v: Top-level FSM with cache and memory coordination
* replacer.v: Logic to replace a specific word within a block
* cache\_controller\_tb.v: Testbench covering all functional cases

We also included a parameters.vh header with cache constants, making the design more configurable.

The FSM handles requests as follows:

1. **IDLE** waits for a valid CPU request.
2. **CHECK\_HIT** scans all 4 blocks in the indexed set.
3. On a **miss**, if the selected block is dirty, it transitions to **EVICT** to write it back.
4. Then it moves to **ALLOCATE**, loads a new block from memory, and returns the result in **SEND\_TO\_CPU**.

Each state is controlled by input readiness (mem\_req\_ready, cpu\_req\_valid) and internal flags (hit, dirty, valid, etc.).

**Implementation Details**

**FSM Breakdown**

Our FSM uses 3-bit encodings and transitions based on combinational logic. We separated control signal assignment from the state transitions, avoiding race conditions during simulation. All state changes are clocked.

localparam IDLE = 3'b000;

localparam CHECK\_HIT = 3'b001;

localparam EVICT = 3'b010;

localparam ALLOCATE = 3'b011;

localparam SEND\_TO\_CPU = 3'b100;

**LRU Replacement**

One of the biggest challenges was implementing an efficient and reliable LRU policy using minimal hardware. We settled on a **2-bit age system**:

* Age 00 = most recently used
* Age 11 = least recently used

Upon access, we update the used block’s age to 00, and increment the others accordingly. This simple approach let us avoid complex timestamp tracking.

if (i != hit\_way && cache[addr\_index][i][536]) begin

if (cache[addr\_index][i][534:533] < cache[addr\_index][hit\_way][534:533]) begin

cache[addr\_index][i][534:533] += 1;

end

end

**Write Support**

We used a dedicated module (replacer.v) to manage partial writes within a 512-bit block. This allowed the controller to update a single word without modifying the rest. The approach used a case statement based on word\_offset:

case (block\_offset)

0: data\_out[31:0] = data\_write;

1: data\_out[63:32] = data\_write;

// ...

15: data\_out[511:480] = data\_write;

endcase

This separation made testing and debugging much easier.

**Testbench and Verification**

Our testbench covered multiple operation types, state transitions, and corner cases. We wrote a task-based interface to send commands and wait for controller responses:

* read\_op(addr)
* write\_op(addr, data)
* wait\_for\_ready()

We then structured the test flow to mimic realistic workloads:

1. Issue a read miss
2. Wait for memory to respond
3. Check the fetched data
4. Issue a write to same line
5. Trigger eviction and verify memory write-back

One major issue we encountered was related to simulation timing. In early versions, some signals (especially mem\_req\_enable) would de-assert too early, resulting in missing memory transactions. We fixed this by adding latching and delay buffers where necessary.

Another challenge was testing the LRU mechanism. We implemented a visualization mechanism using $display inside the testbench to track LRU counters.

A computer screen shot of blue text

AI-generated content may be incorrect.

**Results and Evaluation**

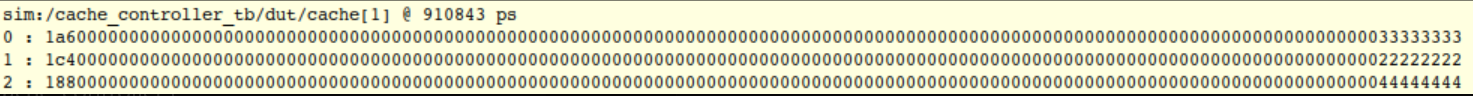
The controller handled directed test cases, covering all corner scenarios:

* Clean and dirty block eviction
* Hits and misses from various index sets
* Writes at multiple word offsets

Here are some representative waveform results:

A screen shot of a computer

AI-generated content may be incorrect.



**Conclusion and Reflections**

This project gave us deep insight into cache memory design, hardware FSM modeling, and debugging multi-cycle logic. Although we began with a solid understanding of associativity and cache layout, implementing everything from scratch including metadata, FSM transitions, partial word writes, and full LRU logic was a major leap.

We faced issues with misaligned state transitions, early signal toggles, and delayed data propagation. By breaking down logic into smaller, testable modules, we solved these problems step-by-step. We also gained experience reading waveforms and using simulation tools like ModelSim more efficiently.