# Cache Controller Project: Final Report

Bodrogean Nadira

Rață Sorin-Gabriel

## 1. Overview of the Design and Implementation Process

The cache controller project was developed to simulate and verify the behavior of a set-associative cache system, focusing on realistic CPU-cache-memory interactions. The design implements a 4-way set-associative cache with 128 sets, 16 words per block (512 bits), and a Least Recently Used (LRU) replacement policy. The controller manages read and write requests from the CPU, handles cache hits and misses, and coordinates with main memory for block allocation and eviction.

The implementation process began with a clear specification of the cache architecture, including parameterization for word size, block size, set count, and associativity. The main modules developed were:

* **cache\_controller.v**: The core finite state machine (FSM) that orchestrates cache operations, manages candidate lines, and interfaces with both the CPU and memory.
* **replacer.v**: A utility module for updating a specific word within a cache block, used during write operations.
* **cache\_controller\_tb.sv**: A comprehensive SystemVerilog testbench that simulates a variety of CPU access patterns, including hits, misses, write-backs, and edge cases.

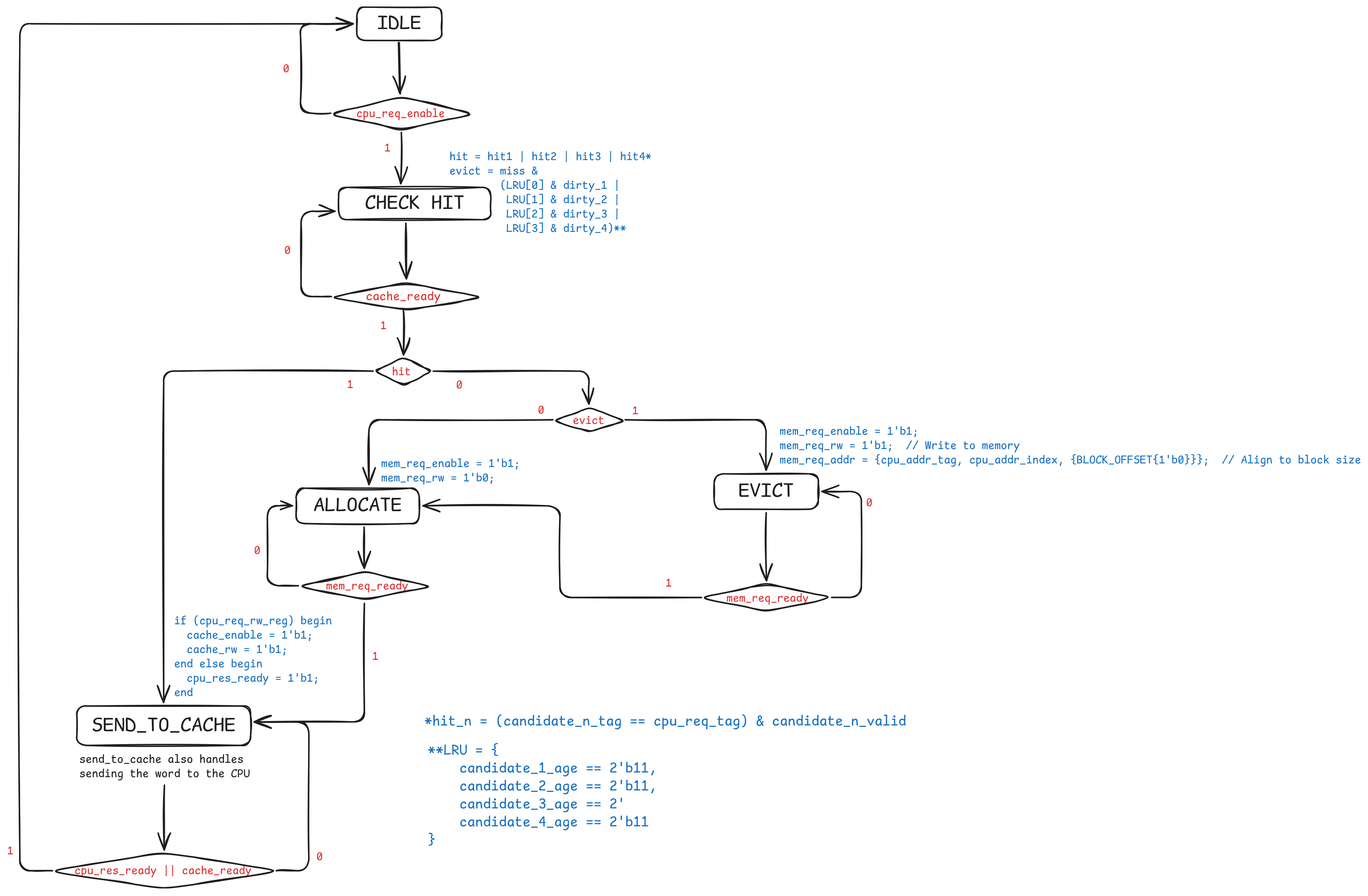
The design emphasizes modularity and parameterization, allowing for easy adaptation to different cache configurations. The FSM in the controller ensures correct sequencing of operations, including hit detection, LRU updates, dirty block eviction, and block allocation from memory.

## FSM and State Transition Diagram

### Finite State Machine (FSM) Overview

The cache controller is implemented as a finite state machine (FSM) that manages the sequence of operations required to process CPU requests, interact with the cache, and coordinate with main memory. The FSM ensures correct handling of cache hits, misses, write-backs, and block allocations.

#### FSM States



* **IDLE**: Waits for a CPU request.
* **CHECK\_HIT**: Checks if the requested address is present in the cache (hit) or not (miss).
* **EVICT**: If a miss occurs and the LRU candidate is dirty, writes the dirty block back to main memory.
* **ALLOCATE**: Allocates a new block from main memory (on miss, after eviction if needed).
* **SEND\_TO\_CACHE**: Updates the cache with new data or returns data to the CPU.

#### State Transition Diagram

Below is an image representation of the FSM:

* From **IDLE**, the FSM waits for a CPU request and transitions to **CHECK\_HIT**.
* In **CHECK\_HIT**, if there is a cache hit, it transitions to **SEND\_TO\_CACHE**. If there is a miss and eviction is needed, it transitions to **EVICT**; otherwise, it goes to **ALLOCATE**.
* **EVICT** waits for the memory to be ready, then transitions to **ALLOCATE**.
* **ALLOCATE** waits for the memory to be ready, then transitions to **SEND\_TO\_CACHE**.
* **SEND\_TO\_CACHE** completes the operation (write to cache or return data to CPU) and returns to **IDLE**.

#### FSM State Encoding in Code

parameter IDLE = 3'b000;  
parameter CHECK\_HIT = 3'b001;  
parameter EVICT = 3'b010;  
parameter ALLOCATE = 3'b011;  
parameter SEND\_TO\_CACHE = 3'b100;  
  
reg [2:0] current\_state, next\_state;  
  
// State transition logic  
always @(\*) begin  
 next\_state = current\_state;  
 case (current\_state)  
 IDLE: begin  
 if (cpu\_req\_enable) begin  
 next\_state = CHECK\_HIT;  
 end  
 end  
 CHECK\_HIT: begin  
 if (cache\_ready\_reg) begin  
 if (hit) begin  
 next\_state = SEND\_TO\_CACHE;  
 end else begin  
 if (evict) begin  
 next\_state = EVICT;  
 end else begin  
 next\_state = ALLOCATE;  
 end  
 end  
 end  
 end  
 EVICT: begin  
 if (mem\_req\_ready) begin  
 next\_state = ALLOCATE;  
 end  
 end  
 ALLOCATE: begin  
 if (mem\_req\_ready) begin  
 next\_state = SEND\_TO\_CACHE;  
 end  
 end  
 SEND\_TO\_CACHE: begin  
 if (cache\_ready || cpu\_res\_ready) begin  
 next\_state = IDLE;  
 end  
 end  
 endcase  
end

## 2. Technical Challenges Encountered and Solutions Implemented

### a. LRU Replacement Policy

**Challenge:** Implementing an efficient and correct LRU policy for a 4-way set-associative cache, ensuring that the “oldest” line is always selected for replacement on a miss.

**Solution:** Each cache line maintains a 2-bit age field. On every access, the controller updates the ages: the accessed line is set to 0, and all valid lines with a lower age are incremented. This logic is implemented combinationally and verified in the testbench. On miss all ages are updated because the block that will be replaced is the oldest one and overflow will correctly make it the youngest. The LRU candidate is selected by finding the line with the maximum age.

**Relevant Code:**

// Age calculation when there is a hit  
// For the accessed candidate: reset to 0  
// For other valid candidates: increment only if their age was less than the hit element's age  
wire [AGE\_BITS-1:0] age\_1\_hit = hit\_1 ? 2'b00 : candidate\_1\_valid ? (candidate\_1\_age < hit\_element\_age ? candidate\_1\_age + 1 : candidate\_1\_age) : candidate\_1\_age;  
wire [AGE\_BITS-1:0] age\_2\_hit = hit\_2 ? 2'b00 : candidate\_2\_valid ? (candidate\_2\_age < hit\_element\_age ? candidate\_2\_age + 1 : candidate\_2\_age) : candidate\_2\_age;  
wire [AGE\_BITS-1:0] age\_3\_hit = hit\_3 ? 2'b00 : candidate\_3\_valid ? (candidate\_3\_age < hit\_element\_age ? candidate\_3\_age + 1 : candidate\_3\_age) : candidate\_3\_age;  
wire [AGE\_BITS-1:0] age\_4\_hit = hit\_4 ? 2'b00 : candidate\_4\_valid ? (candidate\_4\_age < hit\_element\_age ? candidate\_4\_age + 1 : candidate\_4\_age) : candidate\_4\_age;  
  
// Age calculation when there is a miss  
// For valid candidates: increment age (allow overflow back to 00)  
// For invalid candidates: keep current age  
wire [AGE\_BITS-1:0] age\_1\_miss = candidate\_1\_valid ? candidate\_1\_age + 1 : candidate\_1\_age;  
wire [AGE\_BITS-1:0] age\_2\_miss = candidate\_2\_valid ? candidate\_2\_age + 1 : candidate\_2\_age;  
wire [AGE\_BITS-1:0] age\_3\_miss = candidate\_3\_valid ? candidate\_3\_age + 1 : candidate\_3\_age;  
wire [AGE\_BITS-1:0] age\_4\_miss = candidate\_4\_valid ? candidate\_4\_age + 1 : candidate\_4\_age;  
  
// Select between hit and miss age calculations based on whether there was a hit  
assign age\_1 = hit ? age\_1\_hit : age\_1\_miss;  
assign age\_2 = hit ? age\_2\_hit : age\_2\_miss;  
assign age\_3 = hit ? age\_3\_hit : age\_3\_miss;  
assign age\_4 = hit ? age\_4\_hit : age\_4\_miss;  
  
// The least recently used (LRU) candidate is the one with the highest age (one hot encoding)  
wire [BANK-1:0] LRU\_candidate;  
assign LRU\_candidate = {  
 (candidate\_4\_reg[AGE\_START+AGE\_BITS-1:AGE\_START] == 2'b11),  
 (candidate\_3\_reg[AGE\_START+AGE\_BITS-1:AGE\_START] == 2'b11),  
 (candidate\_2\_reg[AGE\_START+AGE\_BITS-1:AGE\_START] == 2'b11),  
 (candidate\_1\_reg[AGE\_START+AGE\_BITS-1:AGE\_START] == 2'b11)  
};

### b. Handling Write-Backs and Dirty Blocks

**Challenge:** Correctly managing dirty blocks during eviction, ensuring that modified data is written back to memory before replacement.

**Solution:** The controller checks the dirty and valid bits of the LRU candidate on a miss. If eviction is required, the block is written to memory before the new block is allocated. The FSM includes explicit EVICT and ALLOCATE states to sequence these operations, and the testbench verifies correct memory transactions.

**Relevant Code:**

wire evict\_1, evict\_2, evict\_3, evict\_4, evict;  
assign evict\_1 = (candidate\_1\_reg[VALID\_BIT\_START] == 1'b1 && candidate\_1\_reg[DIRTY\_BIT\_START] == 1'b1);  
assign evict\_2 = (candidate\_2\_reg[VALID\_BIT\_START] == 1'b1 && candidate\_2\_reg[DIRTY\_BIT\_START] == 1'b1);  
assign evict\_3 = (candidate\_3\_reg[VALID\_BIT\_START] == 1'b1 && candidate\_3\_reg[DIRTY\_BIT\_START] == 1'b1);  
assign evict\_4 = (candidate\_4\_reg[VALID\_BIT\_START] == 1'b1 && candidate\_4\_reg[DIRTY\_BIT\_START] == 1'b1);  
  
// If there is a cache MISS and the LRU candidate is dirty, we need to evict it  
assign evict = miss && (  
 (LRU\_candidate[0] && evict\_1) ||  
 (LRU\_candidate[1] && evict\_2) ||  
 (LRU\_candidate[2] && evict\_3) ||  
 (LRU\_candidate[3] && evict\_4)  
);  
  
// Send the evicted block to main memory  
assign mem\_req\_dataout = evict ? (  
 LRU\_candidate[0] ? candidate\_1\_reg[BLOCK\_DATA\_WIDTH-1:0] :  
 LRU\_candidate[1] ? candidate\_2\_reg[BLOCK\_DATA\_WIDTH-1:0] :  
 LRU\_candidate[2] ? candidate\_3\_reg[BLOCK\_DATA\_WIDTH-1:0] :  
 LRU\_candidate[3] ? candidate\_4\_reg[BLOCK\_DATA\_WIDTH-1:0] : 32'd0  
) : 32'dz;  
  
// State machine transitions for eviction and allocation  
parameter EVICT = 3'b010;  
parameter ALLOCATE = 3'b011;  
  
always @(\*) begin  
 // ...  
 case (current\_state)  
 // ...  
 CHECK\_HIT: begin  
 if (cache\_ready\_reg) begin  
 if (hit) begin  
 next\_state = SEND\_TO\_CACHE; // Cache hit, send data to CPU or write to cache  
 end else begin  
 // Miss: check if we need to evict  
 if (evict) begin  
 next\_state = EVICT; // Need to evict before allocating  
 end else begin  
 next\_state = ALLOCATE; // No eviction needed  
 end  
 end  
 end  
 end  
  
 EVICT: begin  
 if (mem\_req\_ready) begin  
 next\_state = ALLOCATE;  
 end  
 end  
  
 ALLOCATE: begin  
 if (mem\_req\_ready) begin  
 next\_state = SEND\_TO\_CACHE;  
 end  
 end  
 // ...  
 endcase  
end

### c. Synchronization and Signal Timing

**Challenge:** Ensuring correct synchronization between the controller, cache, and memory, especially with respect to ready/enable handshakes and clocking.

**Solution:** The design uses registered signals and flip-flop modules to synchronize candidate data and control signals. The testbench provides realistic clocking and ready/enable pulses, and the controller FSM waits for appropriate ready signals before proceeding to the next state.

**Relevant Code:**

// Registered candidates as registers  
wire [VALID\_BIT + DIRTY\_BIT + AGE\_BITS + TAG\_BITS + BLOCK\_DATA\_WIDTH - 1:0]  
 candidate\_1\_reg, candidate\_2\_reg, candidate\_3\_reg, candidate\_4\_reg;  
  
// Register candidate data when cache\_ready is active  
flipflop\_d #(  
 .WIDTH(VALID\_BIT + DIRTY\_BIT + AGE\_BITS + TAG\_BITS + BLOCK\_DATA\_WIDTH)  
) candidate\_1\_reg\_inst (  
 .clk(clk),  
 .rst\_n(rst\_n),  
 .load(cache\_ready & ~cache\_rw), // Only load when cache\_ready is active  
 .d(candidate\_1),  
 .q(candidate\_1\_reg)  
);  
// ... (similar for candidate\_2\_reg\_inst, candidate\_3\_reg\_inst, candidate\_4\_reg\_inst)

### d. Parameterization and Modularity

**Challenge:** Making the design flexible and reusable for different cache sizes and associativities.

**Solution:** All key parameters (word size, block size, set count, associativity, etc.) are defined as module parameters. The replacer and block\_selector modules are also parameterized, supporting easy scaling and adaptation.

**Relevant Code:**

module cache\_controller #(  
 parameter WORD\_SIZE = 32, // 32 bits per word  
 parameter BLOCK\_OFFSET = 4, // 4 bits for block offset (16 words per block)  
 parameter SETS = 128, // 128 sets in one bank  
 parameter SETS\_BITS = 7, // log2(128) = 7 bits for set index  
 parameter AGE\_BITS = 2, // 2 bits to represent oldest among 4 candidates  
 parameter TAG\_BITS = 21, // 21 bits for tag (32 - BLOCK\_OFFSET - log2(SETS))  
 parameter BLOCK\_DATA\_WIDTH = 512, // 512 bits for data (64 bytes per block)  
 parameter DIRTY\_BIT = 1, // 1 bit for dirty flag,  
 parameter VALID\_BIT = 1, // 1 bit for valid flag  
 parameter BANK = 4 // 4 banks  
) (  
 // ...  
);  
  
module replacer #(  
 parameter WORD\_SIZE = 32,  
 parameter BLOCK\_SIZE = 512,  
 parameter NUM\_SEGMENTS = 16,  
 parameter NUM\_SEGMENTS\_LOG = 4  
) (  
 // ...  
);

### e. Comprehensive Verification

**Challenge:** Creating a testbench that covers all relevant scenarios, including hits, misses, write-backs, and edge cases (e.g., empty candidates).

**Solution:** The SystemVerilog testbench (cache\_controller\_tb.sv) includes tasks for read/write requests, candidate provisioning, and memory response simulation. It runs a suite of test cases covering read/write hits, misses with and without eviction, and operations with empty or partially filled sets. The testbench also generates VCD waveforms for detailed analysis.

**Relevant Code:**

// Task to apply a CPU read request  
task cpu\_read(input [WORD\_SIZE-1:0] addr);  
 cpu\_req\_enable = 1;  
 cpu\_req\_rw = 0;  
 cpu\_req\_addr = addr;  
 @(posedge clk);  
 cpu\_req\_enable = 0;  
 $display("CPU READ request for address 0x%08x. Waiting for response...", addr);  
endtask  
  
// Task to apply a CPU write request  
task cpu\_write(input [WORD\_SIZE-1:0] addr, input [WORD\_SIZE-1:0] data);  
 cpu\_req\_enable = 1;  
 cpu\_req\_rw = 1;  
 cpu\_req\_addr = addr;  
 cpu\_req\_datain = data;  
 @(posedge clk);  
 cpu\_req\_enable = 0;  
 cpu\_req\_rw = 0;  
 $display("CPU WRITE request: address 0x%08x, data 0x%08x", addr, data);  
endtask  
  
// Task to provide cache candidates with specific data  
task provide\_candidates(  
 input [VALID\_BIT + DIRTY\_BIT + AGE\_BITS + TAG\_BITS + BLOCK\_DATA\_WIDTH - 1:0] \_candidate1,  
 \_candidate2, \_candidate3, \_candidate4);  
 candidate\_1 = \_candidate1;  
 candidate\_2 = \_candidate2;  
 candidate\_3 = \_candidate3;  
 candidate\_4 = \_candidate4;  
endtask  
  
// Task to wait for memory request to be asserted  
task wait\_for\_mem\_req();  
 wait (mem\_req\_enable);  
 $display("Memory request asserted at time %0t", $time);  
 mem\_req\_ready = 1; // Indicate memory has valid data  
 @(posedge clk);  
 mem\_req\_ready = 0;  
endtask  
  
// Task to wait for cache access to complete  
task wait\_for\_cache\_access();  
 wait (cache\_ready);  
 $display("Cache access completed at time %0t", $time);  
endtask  
  
// Example test case  
initial begin  
 // ...  
 $display("\nTest Case 1: Read hit in candidate 1");  
 provide\_candidates({1'b1, 1'b1, 2'b10, {9'd0, 12'hABC}, test\_block\_data\_candidates}, {  
 1'b1, 1'b1, 2'b01, {9'd0, 12'hDEF}, test\_block\_data\_candidates}, {  
 1'b1, 1'b1, 2'b00, {9'd0, 12'h123}, test\_block\_data\_candidates}, {  
 1'b1, 1'b1, 2'b11, {9'd0, 12'h456}, test\_block\_data\_candidates});  
 @(posedge clk);  
 cpu\_read({{9'd0}, {12'hABC}, {7'd0}, {4'd0}}); // This should hit in candidate 1  
 wait\_for\_cache\_access();  
 wait (cpu\_res\_ready);  
 $display("Response data: 0x%08x, new age: %b, %b, %b, %b", cpu\_res\_dataout, age\_1, age\_2,  
 age\_3, age\_4);  
 wait (uut.current\_state == uut.IDLE);  
 // ...  
end

## f. Code Structure and Explanation

### 2.f.1. cache\_controller.v — The Main Controller

This is the heart of the project. It implements a finite state machine (FSM) to manage the cache’s behavior, including hit/miss detection, LRU management, and memory interactions.

**Key Features:**

* Parameterized for word size, block size, associativity, and more.
* Receives CPU requests and determines if they are cache hits or misses.
* On a miss, checks if eviction is needed (dirty block) and handles write-back.
* Allocates new blocks from memory and updates the cache.
* Manages LRU ages for all candidates.

module cache\_controller #(  
 parameter WORD\_SIZE = 32, // 32 bits per word  
 parameter BLOCK\_OFFSET = 4, // 4 bits for block offset (16 words per block)  
 parameter SETS = 128, // 128 sets in one bank  
 parameter SETS\_BITS = 7, // log2(128) = 7 bits for set index  
 parameter AGE\_BITS = 2, // 2 bits to represent oldest among 4 candidates  
 parameter TAG\_BITS = 21, // 21 bits for tag (32 - BLOCK\_OFFSET - log2(SETS))  
 parameter BLOCK\_DATA\_WIDTH = 512, // 512 bits for data (64 bytes per block)  
 parameter DIRTY\_BIT = 1, // 1 bit for dirty flag,  
 parameter VALID\_BIT = 1, // 1 bit for valid flag  
 parameter BANK = 4 // 4 banks  
) (  
 input clk,  
 input rst\_n,  
  
 // CPU to cache controller signals  
 input [WORD\_SIZE-1:0] cpu\_req\_addr, // 1 word address  
 input [WORD\_SIZE-1:0] cpu\_req\_datain, // 1 word data input to write  
 output [WORD\_SIZE-1:0] cpu\_res\_dataout, // 1 word response data output to cpu  
 output reg cpu\_res\_ready,  
 input cpu\_req\_rw, // r = 0, w = 1  
 input cpu\_req\_enable,  
  
 // Cache controller to main memory signals  
 output reg [WORD\_SIZE-1:0] mem\_req\_addr, // BLOCK\_OFFSET bits should be always 0 to align to 16 bytes  
 output [BLOCK\_DATA\_WIDTH-1:0] mem\_req\_dataout, // the 64 byte block to be written to main memory (on write back)  
 input [BLOCK\_DATA\_WIDTH-1:0] mem\_req\_datain, // the 64 byte block extracted from main memory (on read miss)  
 output reg mem\_req\_rw, // r = 0, w = 1  
 output reg mem\_req\_enable, // when reading/writing to main memory do not forget to activate  
  
 input mem\_req\_ready, // main memory has valid data at mem\_req\_dataout  
  
 // Physical cache to cache controller signals  
 output reg cache\_enable, // indicates that the cache should do a write/read  
 output reg cache\_rw, // r = 0, w = 1,  
 input cache\_ready, // indicates that the cache has valid data at candidates  
  
 input [VALID\_BIT + DIRTY\_BIT + AGE\_BITS + TAG\_BITS + BLOCK\_DATA\_WIDTH - 1:0] candidate\_1, // candidate from cache line 1  
 input [VALID\_BIT + DIRTY\_BIT + AGE\_BITS + TAG\_BITS + BLOCK\_DATA\_WIDTH - 1:0] candidate\_2, // candidate from cache line 2  
 input [VALID\_BIT + DIRTY\_BIT + AGE\_BITS + TAG\_BITS + BLOCK\_DATA\_WIDTH - 1:0] candidate\_3, // candidate from cache line 3  
 input [VALID\_BIT + DIRTY\_BIT + AGE\_BITS + TAG\_BITS + BLOCK\_DATA\_WIDTH - 1:0] candidate\_4, // candidate from cache line 4  
  
 // assign CACHE\_BANKS[0][INDEX][AGE\_BITS\_START + AGE\_BITS - 1:AGE\_BITS\_START] = age\_1 (when cache\_enable = 1)  
 // assign CACHE\_BANKS[1][INDEX][AGE\_BITS\_START + AGE\_BITS - 1:AGE\_BITS\_START] = age\_2 (when cache\_enable = 1)  
 // assign CACHE\_BANKS[2][INDEX][AGE\_BITS\_START + AGE\_BITS - 1:AGE\_BITS\_START] = age\_3 (when cache\_enable = 1)  
 // assign CACHE\_BANKS[3][INDEX][AGE\_BITS\_START + AGE\_BITS - 1:AGE\_BITS\_START] = age\_4 (when cache\_enable = 1)  
 output [AGE\_BITS-1:0] age\_1,  
 output [AGE\_BITS-1:0] age\_2,  
 output [AGE\_BITS-1:0] age\_3,  
 output [AGE\_BITS-1:0] age\_4,  
  
 output [VALID\_BIT + DIRTY\_BIT + AGE\_BITS + TAG\_BITS + BLOCK\_DATA\_WIDTH - 1:0] candidate\_write, // data to be written to the cache line when hit occurs  
 output [BANK-1:0] bank\_selector // one hot encoding of the bank the candidate\_write  
);  
/// ...  
endmodule

**Explanation:**

* The FSM transitions through states: IDLE, CHECK\_HIT, EVICT, ALLOCATE, SEND\_TO\_CACHE.
* On a CPU request, the controller checks for a hit among the four candidates.
* If a miss and the LRU candidate is dirty, it writes back to memory before allocation.
* LRU ages are updated on every access, ensuring the oldest line is replaced on a miss.

### 2.f.2. replacer.v — Block Word Replacement Utility

This module is used to update a specific word within a cache block, which is essential for write operations (both on hit and miss).

module replacer #(  
 parameter WORD\_SIZE = 32,  
 parameter BLOCK\_SIZE = 512,  
 parameter NUM\_SEGMENTS = 16,  
 parameter NUM\_SEGMENTS\_LOG = 4  
) (  
 input wire [BLOCK\_SIZE-1:0] data\_in,  
 input wire [NUM\_SEGMENTS\_LOG-1:0] block\_offset,  
 input wire [WORD\_SIZE-1:0] data\_write,  
 input wire enable,  
 output reg [BLOCK\_SIZE-1:0] data\_out  
);  
 always @(\*) begin  
 data\_out = data\_in;  
 if (enable) begin  
 case (block\_offset)  
 0: data\_out[0\*WORD\_SIZE+:WORD\_SIZE] = data\_write;  
 1: data\_out[1\*WORD\_SIZE+:WORD\_SIZE] = data\_write;  
 // ... up to 15  
 15: data\_out[15\*WORD\_SIZE+:WORD\_SIZE] = data\_write;  
 default: ;  
 endcase  
 end  
 end  
endmodule

**Explanation:**

* The replacer module takes a block of data and overwrites the word at the specified offset with new data.
* Used for both write hits (updating a word in a cached block) and write misses (updating a word in a block fetched from memory).

### 2.f.3. cache\_controller\_tb.sv — Testbench

This SystemVerilog testbench simulates a variety of scenarios to verify the cache controller’s correctness and provides detailed, human-readable output for each test case. When you run the testbench, you will see output similar to the following:

VCD info: dumpfile cache\_controller\_tb.vcd opened for output.  
Time: 0 | State: 000 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 0 | miss: 1  
  
Test Case 1: Read hit in candidate 1  
Time: 45000 | State: 000 | cache\_enable: 1 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 0 | miss: 1  
CPU READ request for address 0x0055e000. Waiting for response...  
Time: 55000 | State: 001 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 0 | miss: 1  
Cache access completed at time 85000  
Time: 95000 | State: 001 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 1 | miss: 0  
Response data: 0xcac8e000, new age: 00, 10, 01, 11  
Time: 105000 | State: 100 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 1 | miss: 0  
Time: 115000 | State: 000 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 1 | miss: 0  
  
Test Case 2: Read miss without eviction  
Time: 135000 | State: 000 | cache\_enable: 1 | mem\_req\_enable: 0 | bank\_selector: 1000 | hit: 0 | miss: 1  
CPU READ request for address 0x000a0000. Waiting for response...  
Time: 145000 | State: 001 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 1000 | hit: 0 | miss: 1  
Time: 185000 | State: 001 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0010 | hit: 0 | miss: 1  
Memory request asserted at time 195000  
Time: 195000 | State: 011 | cache\_enable: 0 | mem\_req\_enable: 1 | bank\_selector: 0010 | hit: 0 | miss: 1  
Time: 205000 | State: 100 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0010 | hit: 0 | miss: 1  
Response data: 0xbad00000, 1000140bad0000fbad0000ebad0000dbad0000cbad0000bbad0000abad00009bad00008bad00007bad00006bad00005bad00004bad00003bad00002bad00001bad00000, bank: 0010, new age: 11, 00, 10, 01  
Time: 215000 | State: 000 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0010 | hit: 0 | miss: 1  
  
Test Case 3: Write hit in candidate 1  
Time: 235000 | State: 000 | cache\_enable: 1 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 1 | miss: 0  
CPU WRITE request: address 0x006f7801, data 0xcafebabe  
Time: 245000 | State: 001 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 1 | miss: 0  
Cache access completed at time 275000  
Time: 295000 | State: 100 | cache\_enable: 1 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 1 | miss: 0  
Write successful, candidate write data: 1800defcac8e00fcac8e00ecac8e00dcac8e00ccac8e00bcac8e00acac8e009cac8e008cac8e007cac8e006cac8e005cac8e004cac8e003cac8e002cafebabecac8e000, new ages: 00, 11, 10, 01  
Time: 345000 | State: 000 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 1 | miss: 0  
  
Test Case 4: Write miss no eviction  
Time: 365000 | State: 000 | cache\_enable: 1 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 1 | miss: 0  
CPU WRITE request: address 0x006f7801, data 0xcafebabe  
Time: 375000 | State: 001 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 1 | miss: 0  
Cache access completed at time 385000  
Time: 395000 | State: 001 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 0 | miss: 1  
Memory request asserted at time 405000  
Time: 405000 | State: 011 | cache\_enable: 0 | mem\_req\_enable: 1 | bank\_selector: 0001 | hit: 0 | miss: 1  
Time: 415000 | State: 100 | cache\_enable: 1 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 0 | miss: 1  
Write successful, candidate write data: 1800defbad0000fbad0000ebad0000dbad0000cbad0000bbad0000abad00009bad00008bad00007bad00006bad00005bad00004bad00003bad00002cafebabebad00000, new ages: 00, 11, 10, 01  
Time: 465000 | State: 000 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 0 | miss: 1  
  
Test Case 5: Write hit with eviction  
Time: 485000 | State: 000 | cache\_enable: 1 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 0 | miss: 1  
CPU WRITE request: address 0x006f7801, data 0xcafebabe  
Time: 495000 | State: 001 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 0 | miss: 1  
Cache access completed at time 505000  
Memory request asserted at time 525000  
Time: 525000 | State: 010 | cache\_enable: 0 | mem\_req\_enable: 1 | bank\_selector: 0001 | hit: 0 | miss: 1  
Memory request asserted at time 535000  
Time: 535000 | State: 011 | cache\_enable: 0 | mem\_req\_enable: 1 | bank\_selector: 0001 | hit: 0 | miss: 1  
Time: 545000 | State: 100 | cache\_enable: 1 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 0 | miss: 1  
Write successful, candidate write data: 1800defbad0000fbad0000ebad0000dbad0000cbad0000bbad0000abad00009bad00008bad00007bad00006bad00005bad00004bad00003bad00002cafebabebad00000, new ages: 00, 11, 10, 01  
Time: 595000 | State: 000 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 0 | miss: 1  
  
Test Case 6: Read miss with eviction  
Time: 615000 | State: 000 | cache\_enable: 1 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 0 | miss: 1  
CPU READ request for address 0x006f7801. Waiting for response...  
Time: 625000 | State: 001 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 0 | miss: 1  
Cache access completed at time 635000  
Memory request asserted at time 655000  
Time: 655000 | State: 010 | cache\_enable: 0 | mem\_req\_enable: 1 | bank\_selector: 0001 | hit: 0 | miss: 1  
Memory request asserted at time 665000  
Time: 665000 | State: 011 | cache\_enable: 0 | mem\_req\_enable: 1 | bank\_selector: 0001 | hit: 0 | miss: 1  
Read successful, CPU data: 0xbad00001, candidate write data: 0x1000defbad0000fbad0000ebad0000dbad0000cbad0000bbad0000abad00009bad00008bad00007bad00006bad00005bad00004bad00003bad00002bad00001bad00000, new ages: 00, 11, 10, 01  
Time: 675000 | State: 100 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 0 | miss: 1  
Time: 685000 | State: 000 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 0 | miss: 1  
  
Test Case 7: Read miss with empty candidates  
Time: 705000 | State: 000 | cache\_enable: 1 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 0 | miss: 1  
CPU READ request for address 0x006f7802. Waiting for response...  
Time: 715000 | State: 001 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0001 | hit: 0 | miss: 1  
Cache access completed at time 745000  
Time: 755000 | State: 001 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0100 | hit: 0 | miss: 1  
Memory request asserted at time 765000  
Time: 765000 | State: 011 | cache\_enable: 0 | mem\_req\_enable: 1 | bank\_selector: 0100 | hit: 0 | miss: 1  
Read successful, CPU data: 0xbad00002, candidate write data: 0x1000defbad0000fbad0000ebad0000dbad0000cbad0000bbad0000abad00009bad00008bad00007bad00006bad00005bad00004bad00003bad00002bad00001bad00000, new ages: 10, 01, 00, 00  
Time: 775000 | State: 100 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0100 | hit: 0 | miss: 1  
Time: 785000 | State: 000 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0100 | hit: 0 | miss: 1  
  
Test Case 8: Write miss with empty candidates  
Time: 805000 | State: 000 | cache\_enable: 1 | mem\_req\_enable: 0 | bank\_selector: 0100 | hit: 0 | miss: 1  
CPU WRITE request: address 0x006f7801, data 0xcafebabe  
Time: 815000 | State: 001 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0100 | hit: 0 | miss: 1  
Cache access completed at time 845000  
Memory request asserted at time 865000  
Time: 865000 | State: 011 | cache\_enable: 0 | mem\_req\_enable: 1 | bank\_selector: 0100 | hit: 0 | miss: 1  
Time: 875000 | State: 100 | cache\_enable: 1 | mem\_req\_enable: 0 | bank\_selector: 0100 | hit: 0 | miss: 1  
Write successful, candidate write data: 1800defbad0000fbad0000ebad0000dbad0000cbad0000bbad0000abad00009bad00008bad00007bad00006bad00005bad00004bad00003bad00002cafebabebad00000, new ages: 10, 01, 00, 00  
  
Testbench completed. Exiting...  
cache\_controller\_tb.sv:337: $finish called at 925000 (1ps)  
Time: 925000 | State: 000 | cache\_enable: 0 | mem\_req\_enable: 0 | bank\_selector: 0100 | hit: 0 | miss: 1

**How to interpret the output:**

* Each test case is clearly labeled (e.g., “Test Case 1: Read hit in candidate 1”).
* The simulation prints the current simulation time, FSM state, cache and memory enable signals, bank selector, and hit/miss status at key points.
* For each CPU request, the testbench prints the address and whether it is a read or write.
* When a cache access or memory transaction completes, the testbench prints the result, including the data returned and the updated age bits for all candidates.

**Understanding the data patterns:**

* Data values like cac8e\_000 are used to fill cache blocks and are chosen to make cache hits easily recognizable in the output. When you see a response data value like 0xcac8e000, it indicates a cache hit, and the word offset inside the block is encoded in the lower bits.
* Data values like bad\_0000 are used to fill memory blocks and are returned on cache misses. The word offset is also encoded in the lower bits, so you can see which word within the block was accessed.
* This pattern makes it easy to visually distinguish between cache hits (returning cac8e\_... data) and misses (returning bad\_... data) in the output.

**Age bits output:**

* After each operation, the testbench prints the new age bits for all four candidates (e.g., new age: 00, 10, 01, 11).
* The age bits represent the LRU (Least Recently Used) state for each candidate in the set. The candidate with age 00 is the most recently used, and the candidate with the highest value (e.g., 11 in a 2-bit age field) is the least recently used and will be replaced on the next miss.
* The testbench output allows you to track how the LRU policy updates after each access.

**Testbench features:**

* Parameterized to match the cache controller.
* Generates clock and reset signals, and provides mock cache candidates and main memory data.
* Using Systemverilog tasks to improve clarity and reduce code duplication when generating cache candidates and CPU requests
* Runs a suite of test cases: read/write hits, misses with/without eviction, and edge cases (including hits/misses with empty cache lines).
* VCD waveform dumping is enabled for detailed analysis in tools like GTKWave.

module cache\_controller\_tb ();  
 // Parameters  
 parameter WORD\_SIZE = 32;  
 parameter BLOCK\_OFFSET = 4;  
 // ...  
  
 // Signals  
 reg clk;  
 reg rst\_n;  
 // ...  
  
 // Instantiate the cache controller  
 cache\_controller #(  
 .WORD\_SIZE(WORD\_SIZE),  
 .BLOCK\_OFFSET(BLOCK\_OFFSET),  
 // ...  
 ) uut (  
 .clk(clk),  
 .rst\_n(rst\_n),  
 // ...  
 );  
  
 always begin  
 // Test case 1: Read hit in candidate 1  
 $display("\nTest Case 1: Read hit in candidate 1");  
 provide\_candidates({1'b1, 1'b1, 2'b10, {9'd0, 12'hABC}, test\_block\_data\_candidates}, {  
 1'b1, 1'b1, 2'b01, {9'd0, 12'hDEF}, test\_block\_data\_candidates}, {  
 1'b1, 1'b1, 2'b00, {9'd0, 12'h123}, test\_block\_data\_candidates}, {  
 1'b1, 1'b1, 2'b11, {9'd0, 12'h456}, test\_block\_data\_candidates});  
 @(posedge clk);  
 cpu\_read({{9'd0}, {12'hABC}, {7'd0}, {4'd0}}); // This should hit in candidate 1  
 wait\_for\_cache\_access();  
 wait (cpu\_res\_ready);  
 $display("Response data: 0x%08x, new age: %b, %b, %b, %b", cpu\_res\_dataout, age\_1, age\_2,  
 age\_3, age\_4);  
 wait (uut.current\_state == uut.IDLE);  
  
 @(posedge clk);  
 @(posedge clk);  
  
 // Test case 2: Read miss without eviction  
 $display("\nTest Case 2: Read miss without eviction");  
 provide\_candidates({1'b1, 1'b0, 2'b10, {9'd0, 12'hDEF}, test\_block\_data\_candidates}, {  
 1'b1, 1'b0, 2'b11, {9'd0, 12'h123}, test\_block\_data\_candidates}, {  
 1'b1, 1'b1, 2'b01, {9'd0, 12'h456}, test\_block\_data\_candidates}, {  
 1'b1, 1'b1, 2'b00, {9'd0, 12'h789}, test\_block\_data\_candidates});  
 cpu\_read(32'h000A\_0000);  
 wait\_for\_mem\_req();  
 wait (uut.current\_state == uut.IDLE);  
 $display("Response data: 0x%08x, %h, bank: %b, new age: %b, %b, %b, %b", cpu\_res\_dataout,  
 uut.candidate\_write, uut.bank\_selector, age\_1, age\_2, age\_3, age\_4);  
  
 @(posedge clk);  
 @(posedge clk);  
  
 // Test Case 3: Write hit in candidate 1  
 $display("\nTest Case 3: Write hit in candidate 1");  
 provide\_candidates({1'b1, 1'b1, 2'b11, {9'd0, 12'hDEF}, test\_block\_data\_candidates}, {  
 1'b1, 1'b1, 2'b10, {9'd0, 12'h123}, test\_block\_data\_candidates}, {  
 1'b1, 1'b1, 2'b01, {9'd0, 12'h456}, test\_block\_data\_candidates}, {  
 1'b1, 1'b1, 2'b00, {9'd0, 12'h789}, test\_block\_data\_candidates});  
 test\_word\_data = 32'hCAFE\_BABE;  
 cpu\_write({{9'd0}, {12'hDEF}, {7'd0}, {4'd1}},  
 test\_word\_data); // This should hit in candidate 3  
 wait\_for\_cache\_access();  
 wait (uut.current\_state == uut.IDLE);  
 $display("Write successful, candidate write data: %h, new ages: %b, %b, %b, %b",  
 uut.candidate\_write, age\_1, age\_2, age\_3, age\_4);  
  
 @(posedge clk);  
 @(posedge clk);  
  
 // Test case 4: Write miss no eviction  
 $display("\nTest Case 4: Write miss no eviction");  
 provide\_candidates({1'b1, 1'b0, 2'b11, {9'd0, 12'h123}, test\_block\_data\_candidates}, {  
 1'b1, 1'b0, 2'b10, {9'd0, 12'h456}, test\_block\_data\_candidates}, {  
 1'b1, 1'b1, 2'b01, {9'd0, 12'h789}, test\_block\_data\_candidates}, {  
 1'b1, 1'b1, 2'b00, {9'd0, 12'hABC}, test\_block\_data\_candidates});  
 test\_word\_data = 32'hCAFE\_BABE;  
 cpu\_write({{9'd0}, {12'hDEF}, {7'd0}, {4'd1}}, test\_word\_data);  
 wait\_for\_cache\_access();  
 wait\_for\_mem\_req();  
 wait (uut.current\_state == uut.IDLE);  
 $display("Write successful, candidate write data: %h, new ages: %b, %b, %b, %b",  
 uut.candidate\_write, age\_1, age\_2, age\_3, age\_4);  
  
 @(posedge clk);  
 @(posedge clk);  
  
  
 // Test case 5: Write hit with eviction  
 $display("\nTest Case 5: Write hit with eviction");  
 provide\_candidates({1'b1, 1'b1, 2'b11, {9'd0, 12'h123}, test\_block\_data\_candidates}, {  
 1'b1, 1'b0, 2'b10, {9'd0, 12'h456}, test\_block\_data\_candidates}, {  
 1'b1, 1'b0, 2'b01, {9'd0, 12'h789}, test\_block\_data\_candidates}, {  
 1'b1, 1'b1, 2'b00, {9'd0, 12'hABC}, test\_block\_data\_candidates});  
 test\_word\_data = 32'hCAFE\_BABE;  
 cpu\_write({{9'd0}, {12'hDEF}, {7'd0}, {4'd1}}, test\_word\_data);  
 wait\_for\_cache\_access();  
 wait\_for\_mem\_req();  
 wait\_for\_mem\_req();  
 wait (uut.current\_state == uut.IDLE);  
 $display("Write successful, candidate write data: %h, new ages: %b, %b, %b, %b",  
 uut.candidate\_write, age\_1, age\_2, age\_3, age\_4);  
  
 @(posedge clk);  
 @(posedge clk);  
  
  
 // Test case 6: Read miss with eviction  
 $display("\nTest Case 6: Read miss with eviction");  
 provide\_candidates({1'b1, 1'b1, 2'b11, {9'd0, 12'h123}, test\_block\_data\_candidates}, {  
 1'b1, 1'b0, 2'b10, {9'd0, 12'h456}, test\_block\_data\_candidates}, {  
 1'b1, 1'b0, 2'b01, {9'd0, 12'h789}, test\_block\_data\_candidates}, {  
 1'b1, 1'b1, 2'b00, {9'd0, 12'hABC}, test\_block\_data\_candidates});  
 cpu\_read({{9'd0}, {12'hDEF}, {7'd0}, {4'd1}});  
 wait\_for\_cache\_access();  
 wait\_for\_mem\_req();  
 wait\_for\_mem\_req();  
 wait (cpu\_res\_ready);  
 $display(  
 "Read successful, CPU data: 0x%h, candidate write data: 0x%h, new ages: %b, %b, %b, %b",  
 cpu\_res\_dataout, uut.candidate\_write, age\_1, age\_2, age\_3, age\_4);  
 wait (uut.current\_state == uut.IDLE);  
  
 @(posedge clk);  
 @(posedge clk);  
  
 // Test case 7: Read miss with empty candidates  
 $display("\nTest Case 7: Read miss with empty candidates");  
 provide\_candidates({1'b1, 1'b1, 2'b01, {9'd0, 12'h123}, test\_block\_data\_candidates}, {  
 1'b1, 1'b0, 2'b00, {9'd0, 12'h0}, {BLOCK\_DATA\_WIDTH{1'b0}}}, {  
 1'b0, 1'b0, 2'b00, {9'd0, 12'h0}, {BLOCK\_DATA\_WIDTH{1'b0}}}, {  
 1'b0, 1'b0, 2'b00, {9'd0, 12'h0}, {BLOCK\_DATA\_WIDTH{1'b0}}});  
 cpu\_read({{9'd0}, {12'hDEF}, {7'd0}, {4'd2}});  
 wait\_for\_cache\_access();  
 wait\_for\_mem\_req();  
 wait (cpu\_res\_ready);  
 $display(  
 "Read successful, CPU data: 0x%h, candidate write data: 0x%h, new ages: %b, %b, %b, %b",  
 cpu\_res\_dataout, uut.candidate\_write, age\_1, age\_2, age\_3, age\_4);  
 wait (uut.current\_state == uut.IDLE);  
  
 @(posedge clk);  
 @(posedge clk);  
  
 // Test case 8: Write miss with empty candidates  
 $display("\nTest Case 8: Write miss with empty candidates");  
 provide\_candidates({1'b1, 1'b1, 2'b01, {9'd0, 12'h123}, test\_block\_data\_candidates}, {  
 1'b1, 1'b0, 2'b00, {9'd0, 12'h777}, {BLOCK\_DATA\_WIDTH{1'b0}}}, {  
 1'b0, 1'b0, 2'b00, {9'd0, 12'h0}, {BLOCK\_DATA\_WIDTH{1'b0}}}, {  
 1'b0, 1'b0, 2'b00, {9'd0, 12'h0}, {BLOCK\_DATA\_WIDTH{1'b0}}});  
 test\_word\_data = 32'hCAFE\_BABE;  
 cpu\_write({{9'd0}, {12'hDEF}, {7'd0}, {4'd1}}, test\_word\_data);  
 wait\_for\_cache\_access();  
 wait\_for\_mem\_req();  
  
 wait (uut.current\_state == uut.IDLE);  
 $display("Write successful, candidate write data: %h, new ages: %b, %b, %b, %b",  
 uut.candidate\_write, age\_1, age\_2, age\_3, age\_4);  
  
 end  
endmodule

## 3. Analysis of Performance Data Collected During Simulations

Simulation results were collected using the testbench, which exercises the cache controller with a variety of access patterns. Key performance metrics include hit rate, miss rate, and the number of memory transactions (reads/writes).

### a. Hit and Miss Behavior

* **Read/Write Hits:** The controller identifies hits in any of the four candidates, updates the LRU ages, and returns data to the CPU with 2 clock latency .
* **Misses Without Eviction:** When a miss occurs and there is an invalid or clear candidate (dirty = 0 or valid = 0), the controller allocates the new block without requiring a write-back, reducing memory traffic.
* **Misses With Eviction:** If all candidates are valid and the LRU candidate is dirty, the controller performs a write-back to memory before allocating the new block. This is correctly sequenced and verified in the testbench.

### b. LRU Policy Effectiveness

Waveform analysis and testbench output confirm that the LRU policy is correctly maintained. After each access, the ages of the candidates are updated as expected, and the oldest line is always selected for replacement. This ensures optimal cache utilization and minimizes unnecessary evictions.

### c. Memory Traffic

The testbench logs show that memory transactions (reads and writes) occur only on misses and evictions, as expected. Write-backs are performed only for dirty blocks, reducing unnecessary memory writes.

### d. Latency and Throughput

* **Cache Hits:** Data is returned to the CPU with low latency, typically within 2 cycles after the request.
* **Cache Misses:** Misses incur additional latency due to memory access and possible eviction, but the FSM ensures correct sequencing and minimal stalling.
* **Throughput:** The controller must return to the IDLE state because we didn’t pipeline the architecture so throughput can be averaged at 2.5 clocks per request.

### e. Edge Case Handling

The testbench includes cases with empty candidates and partially filled sets. The controller correctly identifies free slots and avoids unnecessary evictions, demonstrating robust handling of all scenarios.

## 4. Conclusion

The cache controller project successfully implements a parameterized, modular, and robust set-associative cache controller with LRU replacement and write-back support. The design addresses key technical challenges, including LRU management, dirty block handling, and synchronization. Comprehensive simulation and waveform analysis confirm correct functionality, efficient memory usage, and robust handling of all edge cases. The project provides a solid foundation for further exploration of cache architectures and performance optimization.