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.

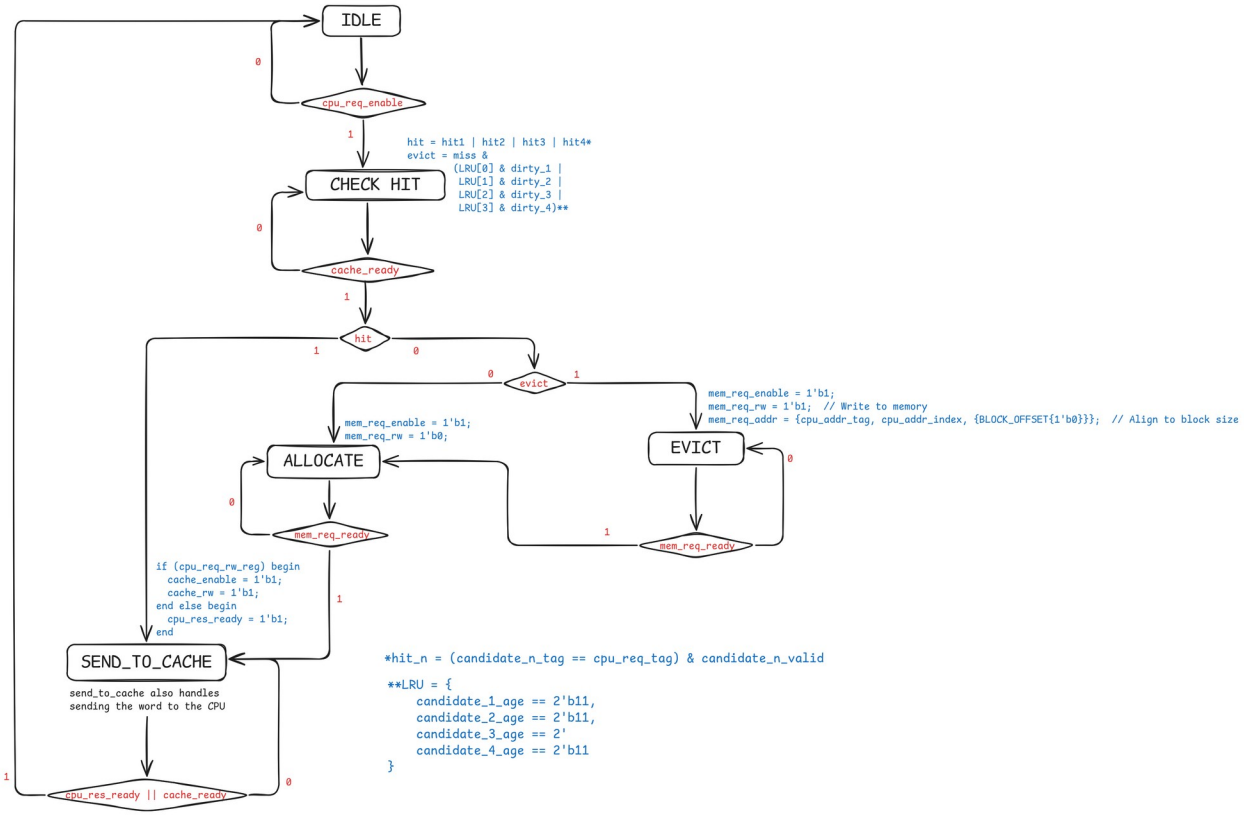
The design emphasizes modularity and parameterization, allowing for easy adaptation to different cache configurations. The FSM in the controller ensures correct sequencing of operations, including hit detection, LRU updates, dirty block eviction, and block allocation from memory.

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.