Politehnica University of Timișoara

Faculty of Automation and Computer Science

CACHE CONTROLLER

Coordinator teacher:

Bozdog Alexandru

Students:

Nemeș Lică

Remețan Mihnea-Florin

Tănase Elena-Alexandra

Tîrsîna Nicoleta

2’nd Semester, 2024-2025

Table of Contents

✓ Project Overview

This project focuses on the **design, simulation, and testing of a cache controller** for a simplified computer system, using a Hardware Description Language (HDL). The project will be carried out by the same student teams who previously developed the ALU module. The cache controller will be constructed using a **finite state machine (FSM)** architecture to manage memory operations and state transitions efficiently.

The objective of the project is to simulate a **4-way set-associative cache controller** that handles memory access requests, implements an **LRU (Least Recently Used)** replacement policy, and follows a **write-back with write-allocate** strategy. Through this exercise, students are expected to deepen their understanding of memory hierarchy, FSM-based hardware design, and HDL programming—skills that are foundational in the field of computer engineering and architecture.

Technical Specifications:

- **Cache Type:** 4-way Set Associative
- **Total Cache Size:** 32 KB
- **Block Size:** 64 Bytes
- **Word Size:** 4 Bytes
- **Number of Sets:** 128
- **Associativity Level:** 4 blocks per set (4-way)
- **Replacement Policy:** Least Recently Used (LRU)
- **Write Policy:** Write-Back with Write Allocate

Simulated Functionalities:

- **Hit/Miss Detection:** The controller checks whether the requested data is available in the cache (hit) or needs to be fetched from main memory (miss).
- **LRU Replacement Policy:** In the event of a miss and a full set, the least recently used block is replaced to make room for new data.
- **Write Handling:** The write-back policy minimizes writes to main memory, while the write-allocate policy ensures that write operations are cached for improved performance.

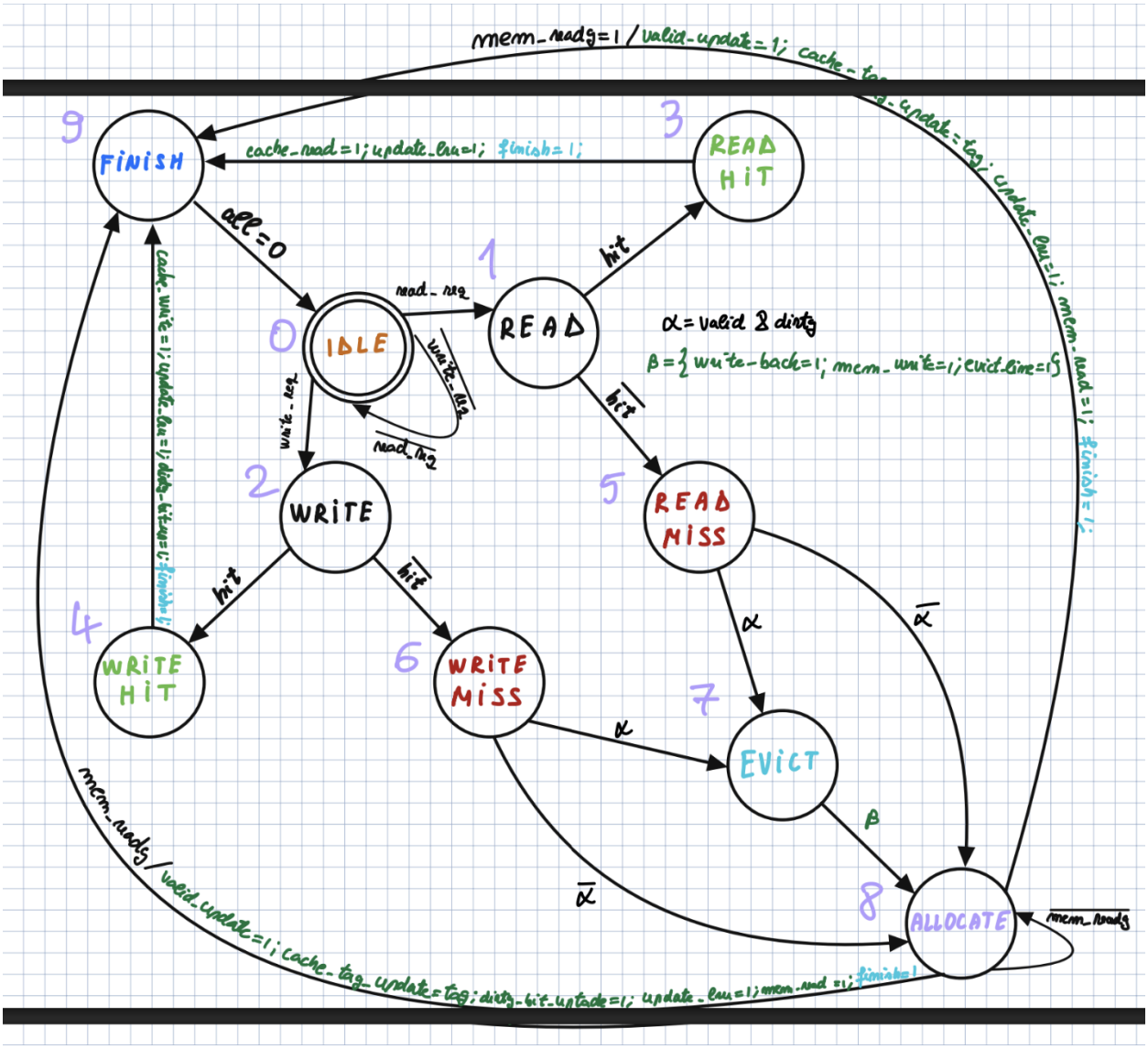
This simulation provides a comprehensive understanding of how a cache controller operates and how cache policies affect system performance, without requiring actual hardware implementation.

✓ Development Environment

- **Visual Studio Code** – for writing and managing Verilog files
- **Simulator**: GTKWave
- **Git** – version control

✓ Project Phases

Design Phase



Relevant inputs:

1. **read_req, write_req** = request for reading/ writing

2. **address** = input address (24 bits)

3. **[511:0] valid,**
   [511:0] dirty_bits,
   [1:0] age [511:0],
   [10:0] cache_tag [511:0],
   ⇨ instead of 4 different banks (4-way Set Associative), we chose to use one big block (every index has 4 blocks) and every block has an input for valid, dirty bit, age and cache tag

4. **mem_ready** = signal indicating if main memory is ready for a write (used during allocation)

Relevant outpus:

1. **cache_read, cache_write** = indicate whether a read or write operation was performed on the cache.

2. **valid_update** = says whether the spot was occupied

3. **[1:0] hit_way** = specifies the index of the cache way that resulted in a hit when the requested data is found

4. **mem_read** = on a cache miss (during allocation), initiates a read from memory.

5. **[10:0] cache_tag_update** = in case of allocate, it updates the cache_tag

6. **dirty_bit_update** = in case of allocate, it updates the dirty_bit

7. **evict_line,**
   write_back,
   mem_write,
   ⇨ evict state, it helps transitioning to the allocate state using Write - Back Write - Allocate principle

8. **lru_way + update_lru** = related to the Least Recently Used (LRU) replacement policy.

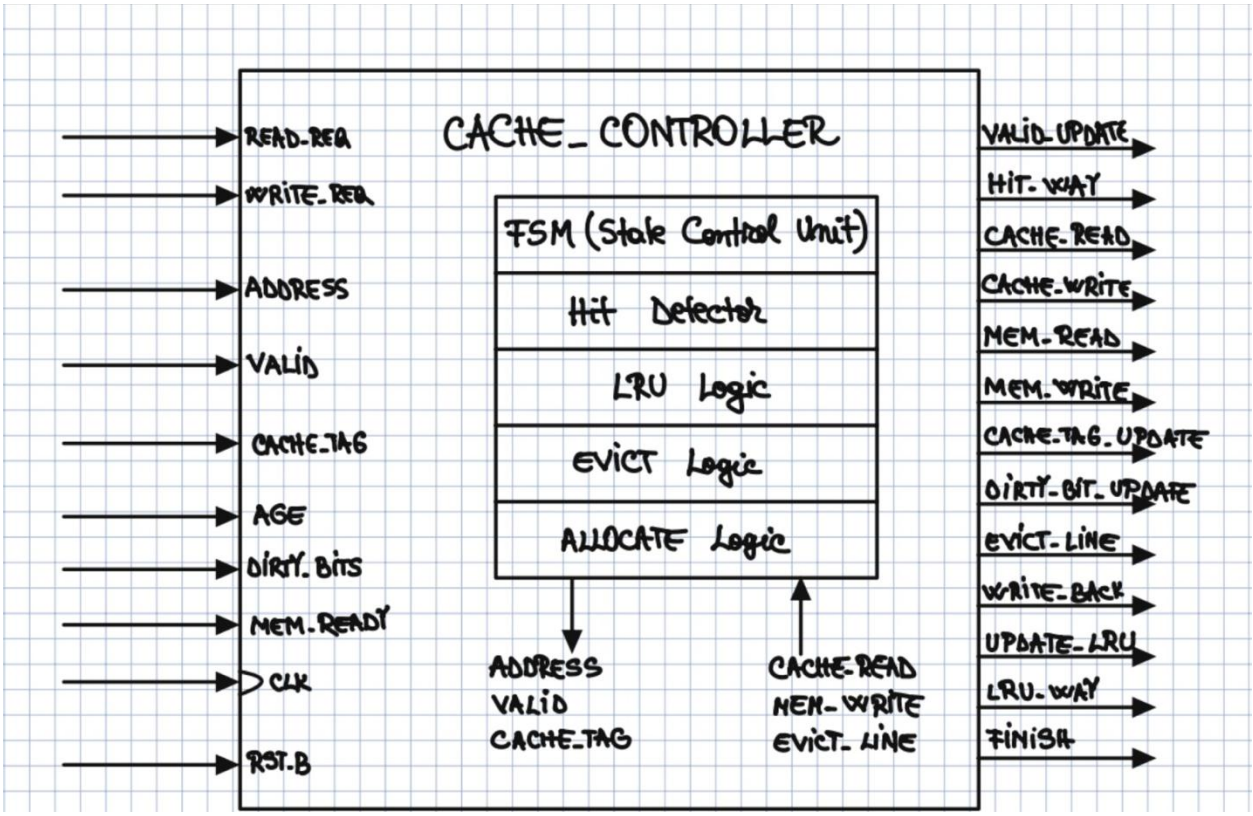
Cache Hit Logic:

The hit is an OR logic between **hit0, hit1, hit2, hit3,** where each hit is an AND logic between the respective valid for the block and the boolean result of whether the tag matches the respective cache_tag

Notes:

We're also using some combinational always blocks for determining the hit_way and lru_way variables, shown in the script code below.

Block Diagram



✅ Code for Cache Controller:

```
module Cache_Controller (

      input clk, rst_b,
      input read_req, write_req,
      input [23:0] address,
      input [511:0] valid,
      input [10:0] cache_tag [511:0],
      input [511:0] dirty_bits,
      input [1:0] age [511:0],
      input mem_ready,

      output reg [1:0]hit_way,
      output reg valid_update,
      output reg [10:0] cache_tag_update,
      output reg dirty_bit_update,

      output reg cache_read, cache_write,
      output reg mem_read,mem_write,
      output reg evict_line, write_back,
      output reg update_lru,
      output reg finish,
      output reg [1:0] lru_way
);

      localparam IDLE = 4'd0;
      localparam READ = 4'd1;
      localparam WRITE = 4'd2;
      localparam READ_HIT = 4'd3;
      localparam WRITE_HIT = 4'd4;
      localparam READ_MISS = 4'd5;
      localparam WRITE_MISS = 4'd6;
      localparam EVICT = 4'd7;
      localparam ALLOCATE = 4'd8;
      localparam FINISH = 4'd9;

      localparam OP_NONE  = 2'b00;
      localparam OP_READ  = 2'b10;
      localparam OP_WRITE = 2'b11;



      reg [3:0] st, st_next;
      reg [1:0] operation;

      wire [10:0] tag = address[23:13];
      wire [6:0] index = address[12:6];
```

```
wire hit0 = valid[index*4+0] && (cache_tag[index*4+0] == tag);
wire hit1 = valid[index*4+1] && (cache_tag[index*4+1] == tag);
wire hit2 = valid[index*4+2] && (cache_tag[index*4+2] == tag);
wire hit3 = valid[index*4+3] && (cache_tag[index*4+3] == tag);

wire hit = hit0 | hit1 | hit2 | hit3;

always @(*)
begin
      if (hit0) hit_way = 2'd0;
      else if (hit1) hit_way = 2'd1;
      else if (hit2) hit_way = 2'd2;
      else if (hit3) hit_way = 2'd3;
      else hit_way = 2'd0;  // default
end

always @(*)
begin
      if (!valid[index*4+0]) lru_way = 2'd0;
      else if (!valid[index*4+1]) lru_way = 2'd1;
      else if (!valid[index*4+2]) lru_way = 2'd2;
      else if (!valid[index*4+3]) lru_way = 2'd3;
      else
      begin
            if (age[index*4+0] >= age[index*4+1] && age[index*4+0]
            >= age[index*4+2] && age[index*4+0] >= age[index*4+3])
            lru_way = 2'd0;

            else if (age[index*4+1] >= age[index*4+0] &&
            age[index*4+1] >= age[index*4+2] && age[index*4+1]
            >= age[index*4+3]) lru_way = 2'd1;

            else if (age[index*4+2] >= age[index*4+0] &&
            age[index*4+2] >= age[index*4+1] && age[index*4+2]
            >= age[index*4+3]) lru_way = 2'd2;

            else lru_way = 2'd3;
      end
end

initial
begin
      hit_way = 2'd0;
      valid_update = 1'd0;
      cache_tag_update = 10'd0;
      dirty_bit_update = 1'd0;

      cache_read = 1'd0; cache_write = 1'd0;
      mem_read = 1'd0; mem_write = 1'd0;
      evict_line = 1'd0; write_back = 1'd0;
      update_lru = 1'd0;
      finish = 1'd0;
```

```
      lru_way = 2'd0;

      st=3'd0;
      st_next=3'd0;
      operation=2'd0;
end

task clear_outputs;
begin
      valid_update = 1'd0;
      cache_tag_update = 10'd0;
      dirty_bit_update = 1'd0;
      cache_read = 1'd0;
      cache_write = 1'd0;
      mem_read = 1'd0;
      mem_write = 1'd0;
      evict_line = 1'd0;
      write_back = 1'd0;
      update_lru = 1'd0;
      operation = OP_NONE;
end
endtask

always @(posedge clk)
begin
      case (st)
      IDLE: begin
            if (read_req)
            begin
                  operation = OP_READ;
                  st_next = READ;
            end

            else if(write_req)
            begin
                  operation = OP_WRITE;
                  st_next = WRITE;
            end

            else finish=0;

      end

      READ: begin
            if (hit)
                  st_next = READ_HIT;

            else
                  st_next = READ_MISS;

      end
```

```
WRITE: begin
      if (hit)
            st_next = WRITE_HIT;

      else
            st_next = WRITE_MISS;
end

READ_HIT: begin
      cache_read = 1;
      update_lru = 1;
      st_next = FINISH;
end

WRITE_HIT: begin
      cache_write = 1;
      update_lru = 1;
      dirty_bit_update=1;
      st_next = FINISH;
end

READ_MISS: begin
      if(valid[index*4 + lru_way] && dirty_bits[index*4 +
      lru_way]) st_next = EVICT;

      else st_next = ALLOCATE;
end

WRITE_MISS: begin
      if(valid[index*4 + lru_way] && dirty_bits[index*4 +
      lru_way]) st_next = EVICT;

      else st_next = ALLOCATE;
end

EVICT: begin
      write_back = 1;
      mem_write = 1;
      evict_line = 1;
      st_next = ALLOCATE;
end

ALLOCATE: begin
      if (mem_ready)
      begin
            write_back = 0;
            mem_write = 0;
            evict_line = 0;
            valid_update = 1;
            cache_tag_update = tag;
            dirty_bit_update = operation[0];
            update_lru=1;
```

```
                mem_read=1;
                st_next = FINISH;
            end
        end

        FINISH: begin
            clear_outputs();
            finish = 1'd1;
            st_next = IDLE;
        end
    endcase
end

always @(posedge clk or negedge rst_b)
begin
        if (!rst_b)
            st <= IDLE;

        else
            st <= st_next;
end

endmodule
```

Each module was implemented in Verilog using **modular and structural design** principles.

✅ Code ExampleTesting Phase

To evaluate the correctness and efficiency of the simulated cache controller, a comprehensive testing phase was conducted. A total of **six distinct scenarios** were designed and executed, each targeting specific aspects of the controller's functionality and behavior.

Scenario 1: IDLE → READ → READ HIT → FINISH → IDLE

Scenario 2: IDLE → READ → READ MISS → ALLOCATE → FINISH → IDLE

Scenario 3: IDLE → READ → READ MISS → EVICT → ALLOCATE → FINISH → IDLE

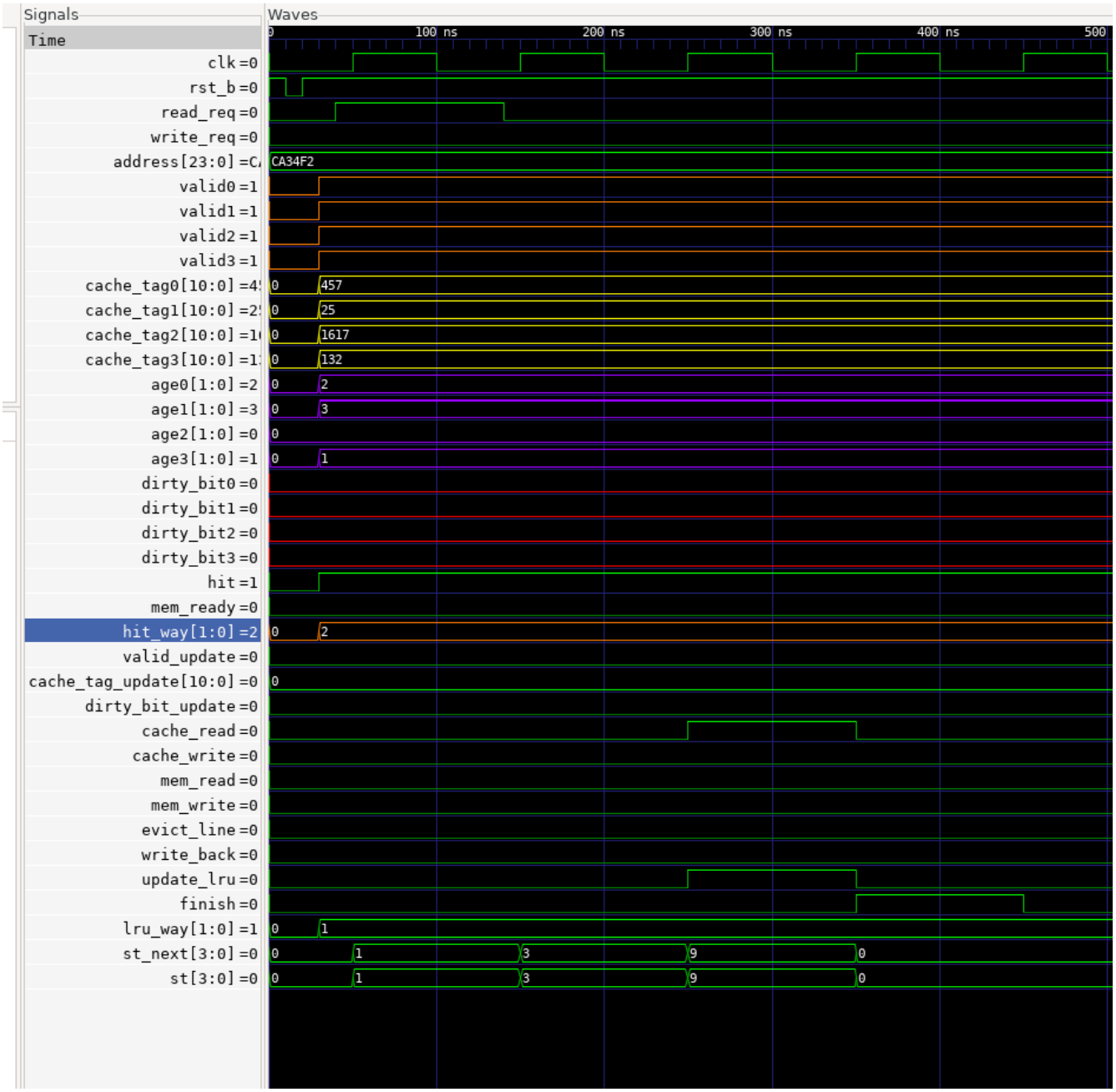
Scenario 4: IDLE → WRITE → WRITE HIT → FINISH → IDLE

Scenario 5: IDLE → WRITE → WRITE MISS → ALLOCATE → FINISH → IDLE

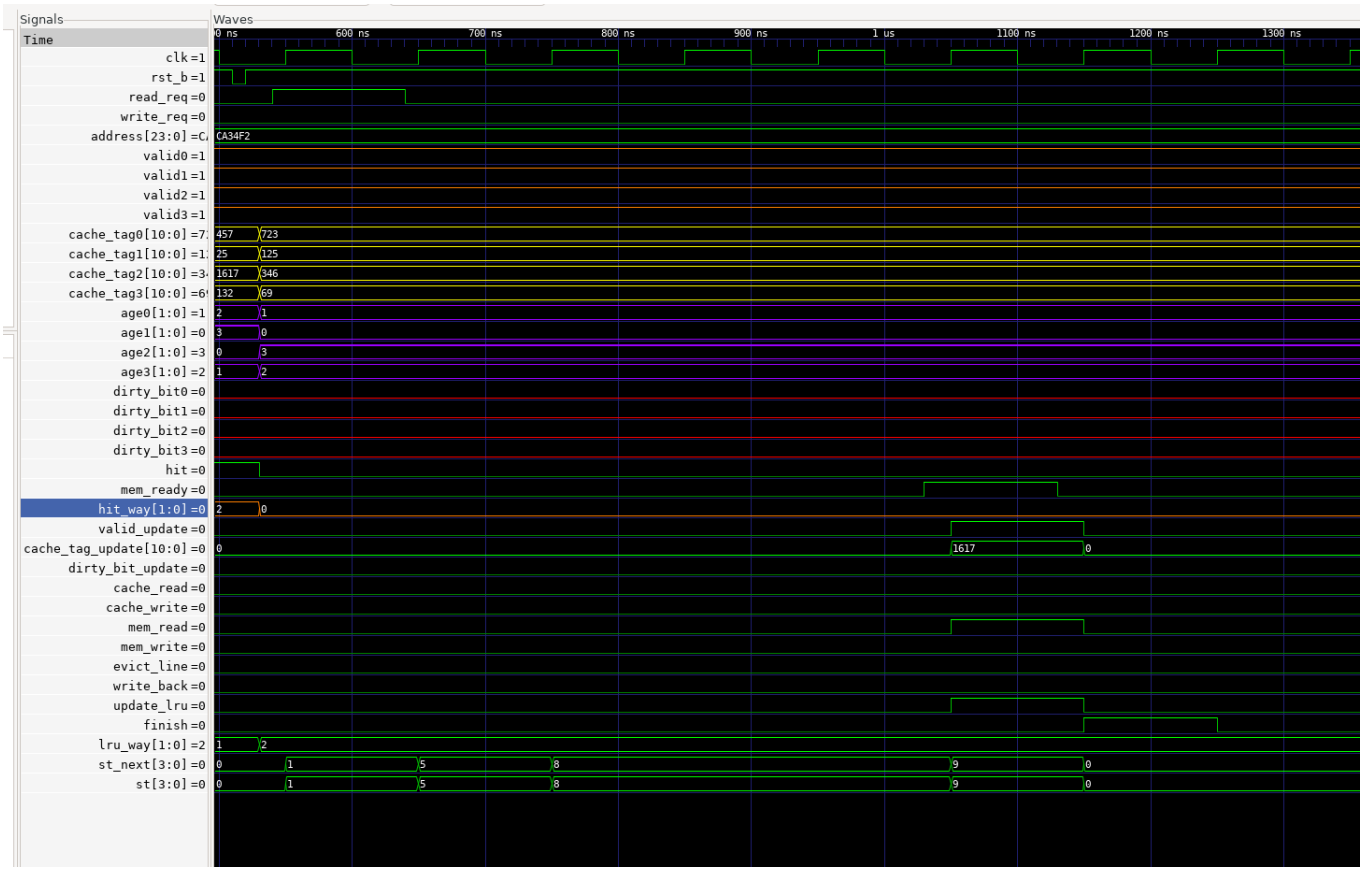
Scenario 6: IDLE → WRITE → WRITE MISS → EVICT → ALLOCATE → FINISH → IDLE

✓ Simulation Output

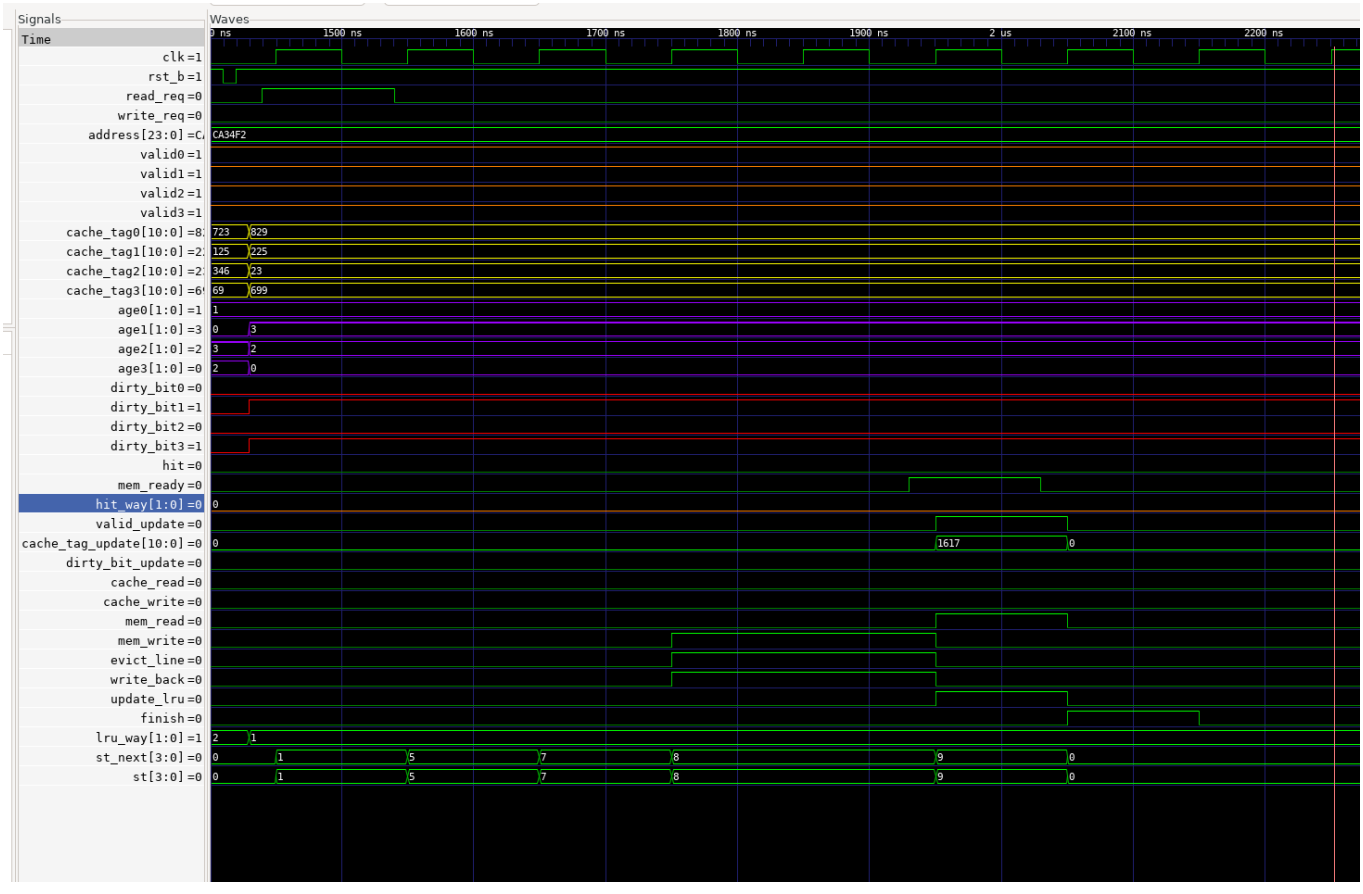
Scenario 1



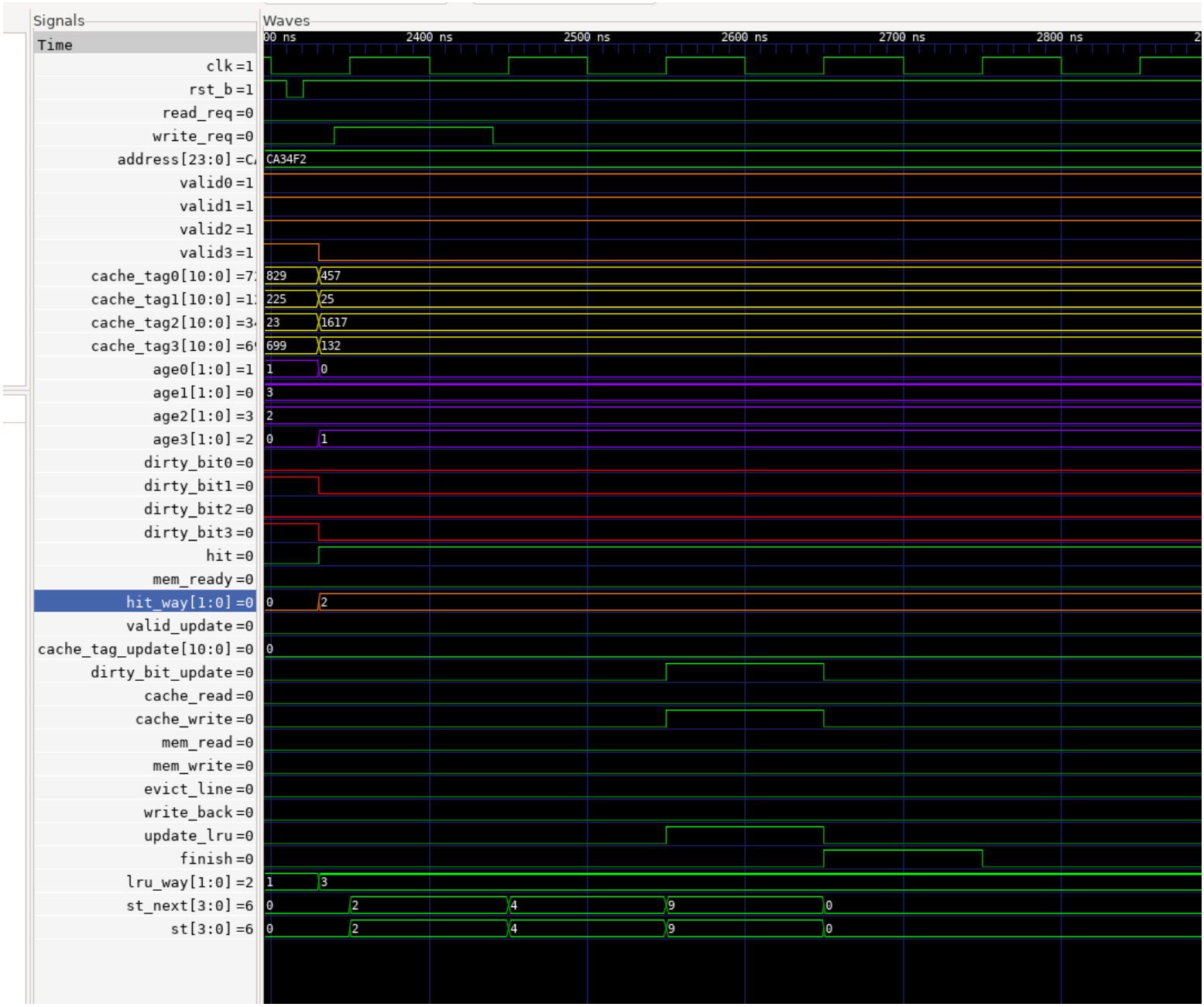
Scenario 2



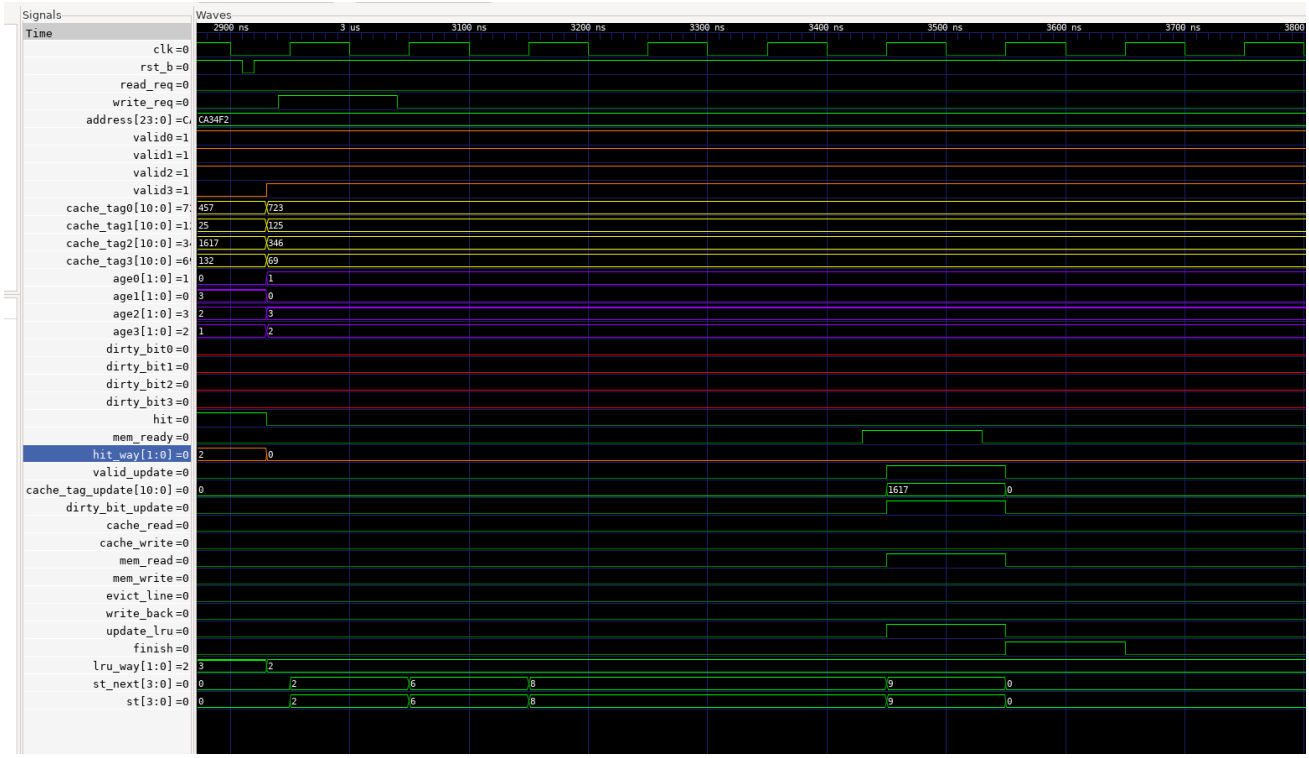
Scenario 3



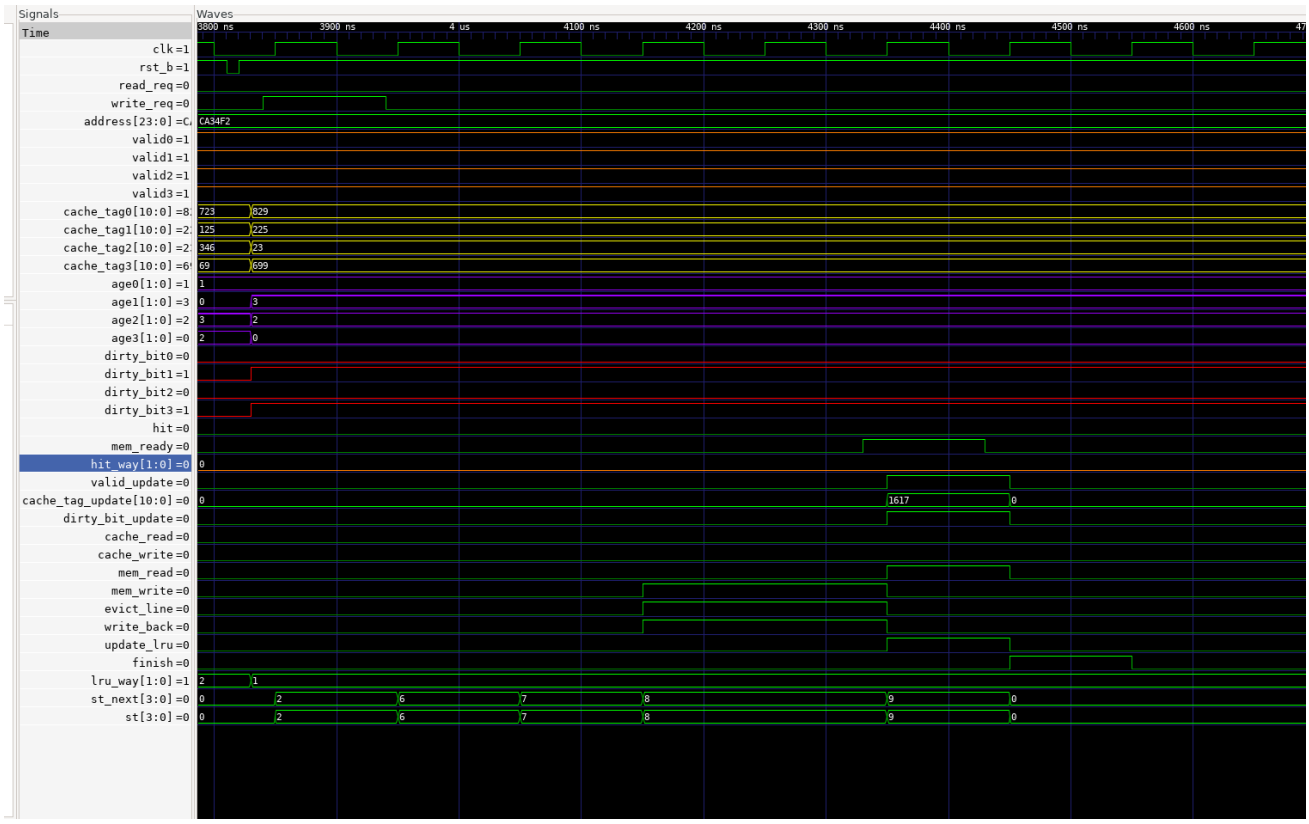
Scenario 4


Signals
Waves
Time
00 ns
2400 ns
2500 ns
2600 ns
2700 ns
2800 ns
clk =1
rst_b =1
read_req =0
write_req =0
address[23:0] =C
CA34F2
valid0 =1
valid1 =1
valid2 =1
valid3 =1
cache_tag0[10:0] =7
829
457
cache_tag1[10:0] =1
225
25
cache_tag2[10:0] =3
23
1617
cache_tag3[10:0] =6
699
132
age0[1:0] =1
1
0
age1[1:0] =0
3
age2[1:0] =3
2
age3[1:0] =2
0
1
dirty_bit0 =0
dirty_bit1 =0
dirty_bit2 =0
dirty_bit3 =0
hit =0
mem_ready =0
hit_way[1:0] =0
0
2
valid_update =0
cache_tag_update[10:0] =0
0
dirty_bit_update =0
cache_read =0
cache_write =0
mem_read =0
mem_write =0
evict_line =0
write_back =0
update_lru =0
finish =0
lru_way[1:0] =2
1
3
st_next[3:0] =6
0
2
4
9
0
st[3:0] =6
0
2
4
9
0

Scenario 5



Scenario 6



✓ Challenges & Resolutions

Challenge	Resolution
We thought we had to implement the memory too, not just the cache controller.	We reread the specifications and asked the lab instructor for clarification.
We were giving the hit as input instead of the address.	We modified the design to calculate the hit from the address.
State transitions were not well-defined in the initial FSM design.	We revised the FSM by clearly defining each state and its transitions based on specific cache events (hit, miss, write-back…), which improved clarity and functionality.

✓ Conclusion

In this project, we successfully designed and implemented a cache controller in Verilog, focusing on cache hit/miss detection, the LRU replacement policy, and the Write-Back with Write-Allocate strategy. We used a Finite State Machine (FSM) architecture to manage the different states and transitions involved in cache operations.

The controller was tested through several scenarios, validating its functionality in handling read and write operations, cache hits, misses, evictions, and allocations. Despite initial challenges, such as misunderstanding the project scope and hit detection method, we resolved them and refined the design.