Facultatea de Automatica si Calculatoare, Universitatea Politehnica Timisoara

DOCUMENTATIE
CACHE CONTROLLER

PROFESOR COORDONATOR:

Bozdog Alexandru

PROIECT REALIZAT DE:

Horvath Maximilian-Mark

Stochean Liviana

Szatmari Levente-Alex

CUPRINS:

• 1. O SCURTA PREZENTARE:

Proiectul implementat simuleaza un ***cache controller*** capabil sa gestioneze operatii de ***citire*** si ***scriere*** intre procesor si memorie, utilizand ***Verilog***.

Cache-ul este de ***tip 4-way set associative***, structurat pe 128 de seturi si blocuri de 64 de bytes.

Controlul este realizat printr-o masina de stari finite (FSM) de tip Moore, care trateaza fiecare cerere in functie de situatie:

->***HIT***

->MISS

->EVICT.

Au fost folosite tehnici precum identificarea de blocuri valide, marcarea cu bit dirty, selectia caii potrivite si actualizarea logica ***LRU***.

Proiectul este structurat in doua componente principale:

->Modulul cache_controller, care contine logica FSM si manipularea datelor din cache

->Testbench-ul, care verifica functionalitatea si performanta prin secvente de comenzi de citire si scriere

Testbench-ul asociat acopera toate scenariile posibile si masoara performanta prin rata de hit si latența medie per operatie.

- ## 2. 4-WAY SA

Cache-ul implementat in acest proiect este de ***tip 4-way set associative***, ceea ce inseamna ca fiecare set din cache poate contine pana la ***4 linii (cai).***

Acest model ofera un echilibru bun intre viteza de acces si rata de utilizare eficienta a spatiului.

In total, cache-ul este organizat in 128 de seturi, iar fiecare set are 4 cai (ways).

Atunci cand procesorul face o cerere ***de citire sau scriere***, adresa este impartita in campuri:

->***tag***

->index

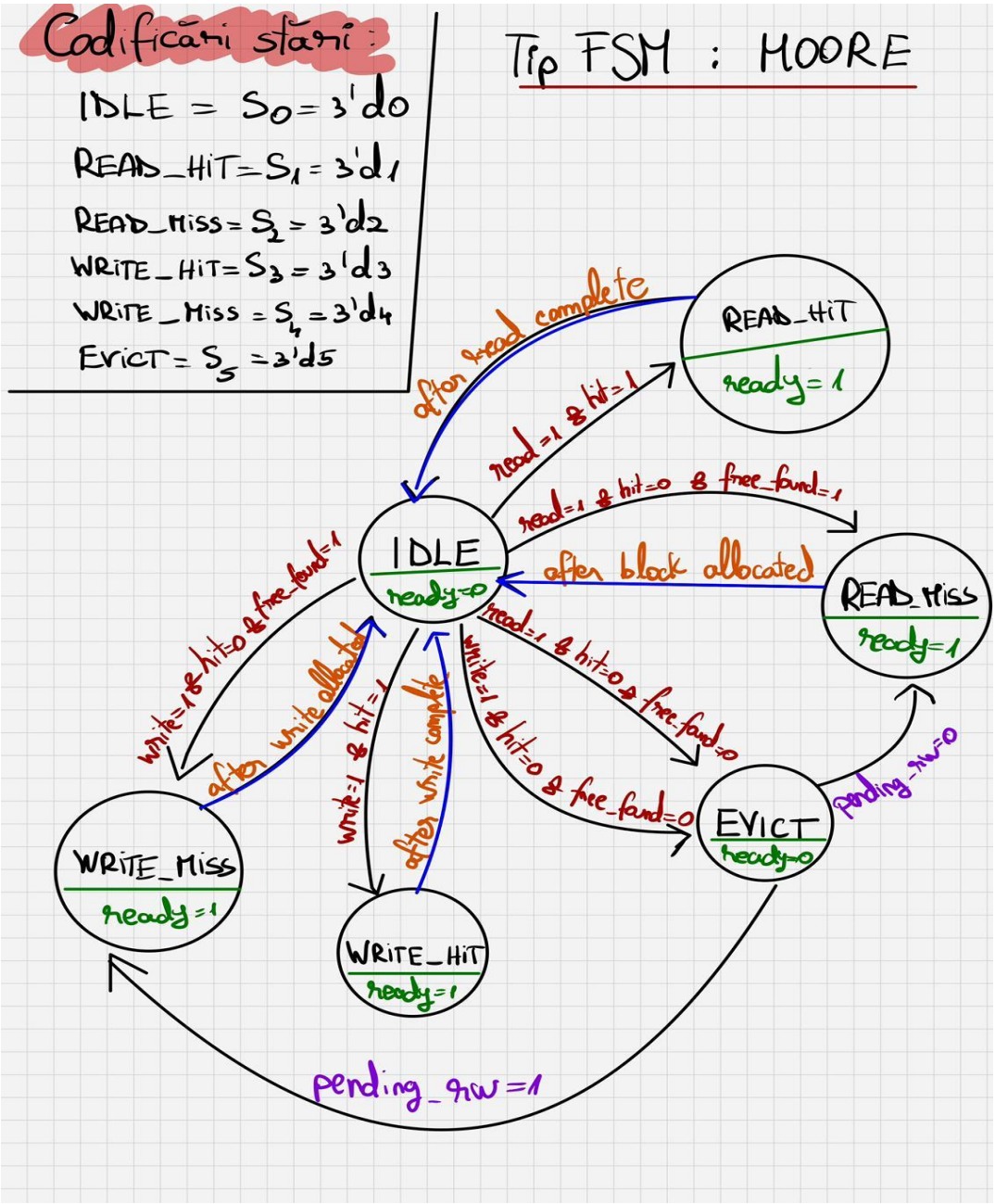
-> offset.

Indexul determina setul tinta, iar controllerul verifica toate cele 4 cai din acel set pentru a identifica un ***hit***.

Daca niciuna dintre cele 4 intrari nu corespunde, controlerul cauta o linie libera (***free_found***) sau alege o linie de inlocuit folosind politica ***LRU (Least Recently Used).***

Aceasta abordare reduce semnificativ conflictele si creste probabilitatea ca datele sa fie deja disponibile in cache.

- # 3. DIAGRAMA DE TIP FSM



Aceasta diagrama reprezinta o vedere de ansamblu a arhitecturii interne a unui controller de cache si modul in care interactioneaza cu memoria principala si procesorul.

Elementele principale prezentate sunt:

- **Semnalele de intrare**: addr, wdata, read, write, clk, reset – provenite de la processor

- **State Logic (FSM)**: controleaza comportamentul cache-ului in functie de comenzi si semnale

- **Cache Memory**:
  - tags: identificatori pentru a verifica daca datele se afla in cache
  - valid: indica daca o linie este valida
  - dirty: indica daca linia a fost modificata
  - data: contine blocurile efective de memorie

- **Indexare**:
  - tags, witid, set_idx, word_c – folosite pentru accesul la structurile interne

- **Interfata cu memoria principala**: in caz de MISS sau EVICT se face acces extern

- **Semnalele de iesire**:
  - rdata: datele returnate catre procesor
  - ready: semnal care indica finalizarea operatiei

Diagrama ajuta la intelegerea fluxului de date si control intre componente si este esentiala pentru proiectarea si verificarea cache-ului.

- # 4. COD:

```
`timescale 1ns / 1ps

module cache_controller(
    input             clk,
    input             reset,
    input             read,
    input             write,
    input      [31:0] addr,
    input      [31:0] wdata,
    output reg [31:0] rdata,
    output reg        ready
);

parameter ADDR_WIDTH  = 32;
parameter DATA_WIDTH  = 32;
parameter SETS        = 128;
parameter WAYS        = 4;
parameter BLOCK_SIZE  = 64;
parameter WORD_BYTES  = 4;
parameter WORDS       = BLOCK_SIZE/WORD_BYTES; // 16 words per block
parameter INDEX_BITS  = 7;
parameter OFFSET_BITS = 6;
parameter TAG_BITS    = ADDR_WIDTH - INDEX_BITS - OFFSET_BITS;
parameter WORD_OFF    = OFFSET_BITS - 2;

localparam IDLE        = 3'd0,
           READ_HIT    = 3'd1,
           READ_MISS   = 3'd2,
           WRITE_HIT   = 3'd3,
           WRITE_MISS  = 3'd4,
           EVICT       = 3'd5;

reg [2:0] state, next_state;

reg [TAG_BITS-1:0]   tags    [0:SETS-1][0:WAYS-1];
reg                  valid   [0:SETS-1][0:WAYS-1];
reg                  dirty   [0:SETS-1][0:WAYS-1];
reg [DATA_WIDTH-1:0] data    [0:SETS-1][0:WAYS-1][0:WORDS-1];
reg [1:0]            lru     [0:SETS-1][0:WAYS-1];
```

```
wire [INDEX_BITS-1:0] set_idx    = addr[OFFSET_BITS +: INDEX_BITS];
wire [TAG_BITS-1:0]   tag_in     = addr[OFFSET_BITS+INDEX_BITS +: TAG_BITS];
wire [WORD_OFF-1:0]   word_off   = addr[OFFSET_BITS-1:2];

reg hit;
reg [1:0] hit_way;
reg free_found;
reg [1:0] free_way;

reg pending_rw;   // 0 = read, 1 = write
reg [1:0] alloc_way;

integer i, j, k;

always @(*) begin
    hit        = 1'b0;
    hit_way    = 2'b00;
    free_found = 1'b0;
    free_way   = 2'b00;
    for (i = 0; i < WAYS; i = i + 1) begin
        if (valid[set_idx][i] && tags[set_idx][i] == tag_in) begin
            hit     = 1'b1;
            hit_way = i[1:0];
        end
        if (!valid[set_idx][i] && !free_found) begin
            free_found = 1'b1;
            free_way   = i[1:0];
        end
    end
end

task update_lru(
    input [INDEX_BITS-1:0] idx,
    input [1:0]           way
);
    reg [1:0] old_val;
    integer   m;
begin
    old_val = lru[idx][way];
    for (m = 0; m < WAYS; m = m + 1) begin
        if (lru[idx][m] < old_val) begin
            lru[idx][m] = lru[idx][m] + 1;
```

```
        end
    end
    lru[idx][way] = 0;
end
endtask

function [1:0] find_lru;
    input [INDEX_BITS-1:0] idx;
    reg   [1:0]            maxv;
    integer                n;
begin
    maxv     = 2'b00;
    find_lru = 2'b00;
    for (n = 0; n < WAYS; n = n + 1) begin
        if (lru[idx][n] > maxv) begin
            maxv     = lru[idx][n];
            find_lru = n[1:0];
        end
    end
end
endfunction

always @(posedge clk or posedge reset) begin
    if (reset) begin
        state <= IDLE;
        ready <= 1'b0;
        for (i = 0; i < SETS; i = i + 1) begin
            for (j = 0; j < WAYS; j = j + 1) begin
                valid[i][j] = 1'b0;
                dirty[i][j] = 1'b0;
                lru[i][j]   = j[1:0];
            end
        end
    end else begin
        state <= next_state;
        if (state == IDLE && (read || write)) begin
            pending_rw  <= write;
            if (hit)
                alloc_way = hit_way;
            else if (free_found)
                alloc_way = free_way;
            else
                alloc_way = find_lru(set_idx);
```

```
        end
    end
end

always @(*) begin
    next_state = state;
    ready      = 1'b0;
    rdata      = 32'b0;

    case (state)
        IDLE: begin
            if (read) begin
                if (hit) begin
                    next_state = READ_HIT;
                end else if (free_found) begin
                    next_state = READ_MISS;
                end else begin
                    next_state = EVICT;
                end
            end else if (write) begin
                if (hit) begin
                    next_state = WRITE_HIT;
                end else if (free_found) begin
                    next_state = WRITE_MISS;
                end else begin
                    next_state = EVICT;
                end
            end
        end

        READ_HIT: begin
            rdata = data[set_idx][alloc_way][word_off];
            update_lru(set_idx, alloc_way);
            ready      = 1'b1;
            next_state = IDLE;
        end

        WRITE_HIT: begin
            data[set_idx][alloc_way][word_off] = wdata;
            dirty[set_idx][alloc_way]          = 1'b1;
            update_lru(set_idx, alloc_way);
            ready      = 1'b1;
            next_state = IDLE;
```

```
        end

        READ_MISS: begin
            tags[set_idx][alloc_way]    = tag_in;
            valid[set_idx][alloc_way]   = 1'b1;
            dirty[set_idx][alloc_way]   = 1'b0;
            for (i = 0; i < WORDS; i = i + 1) begin
                data[set_idx][alloc_way][i] = 32'b0;
            end
            rdata      = 32'b0;
            update_lru(set_idx, alloc_way);
            ready      = 1'b1;
            next_state = IDLE;
        end

        WRITE_MISS: begin
            tags[set_idx][alloc_way]    = tag_in;
            valid[set_idx][alloc_way]   = 1'b1;
            dirty[set_idx][alloc_way]   = 1'b1;
            for (i = 0; i < WORDS; i = i + 1) begin
                data[set_idx][alloc_way][i] = 32'b0;
            end
            data[set_idx][alloc_way][word_off] = wdata;
            update_lru(set_idx, alloc_way);
            ready      = 1'b1;
            next_state = IDLE;
        end

        EVICT: begin
            valid[set_idx][alloc_way] = 1'b0;
            dirty[set_idx][alloc_way] = 1'b0;
            if (pending_rw)
                next_state = WRITE_MISS;
            else
                next_state = READ_MISS;
        end

        default: begin
            next_state = IDLE;
        end
    endcase
end
endmodule
```

Designul controllerului de cache este realizat modular, folosind Verilog HDL. Arhitectura sa este bazata pe o masina de stari finite (FSM) de tip Moore, care raspunde in mod determinist la semnalele de intrare, tranzitionand prin starile definite pentru fiecare operatie:

-> IDLE

-> READ_HIT

-> READ_MISS

-> WRITE_HIT

-> WRITE_MISS

-> EVICT.

Controllerul gestioneaza memoria cache organizata in 128 de seturi, fiecare cu 4 cai (4-way set associative). Pentru fiecare operatie, adresa primita este decodificata in trei componente:

-> OFFSET

-> INDEX

-> TAG.

INDEX-ul selecteaza setul, iar TAG-ul este comparat cu intrarile existente pentru a detecta un HIT.

Blocul combinational de detectie verifica paralel cele 4 cai pentru a stabili daca datele sunt deja prezente (hit) sau daca exista o linie libera (free_found). Daca nu exista, se foloseste politica LRU pentru a determina linia ce va fi inlocuita. In caz de miss si dirty, are loc o operatie de EVICT, urmata de scrierea noului bloc.

In plus, controllerul include structuri pentru validare, bitul dirty si actualizarea automata a informatiilor LRU in functie de accese. Semnalul `ready` este activat doar dupa completarea cu succes a unei operatii.

Codul este organizat pe mai multe blocuri `always`, fiecare avand rol specific:

-> logica combinationala (detectii)

-> logica secventiala (actualizari de stare)

-> outputuri sincronizate.

- **5. TB:**

```
`timescale 1ns / 1ps

module cache_controller_tb;
    reg clk;
    reg reset;

    reg         read;
    reg         write;
    reg  [31:0] addr;
    reg  [31:0] wdata;
    wire [31:0] rdata;
    wire        ready;

    cache_controller uut (
        .clk   (clk),
        .reset (reset),
        .read  (read),
        .write (write),
        .addr  (addr),
        .wdata (wdata),
        .rdata (rdata),
        .ready (ready)
    );

    wire [2:0] fsm_state = uut.state;

    initial clk = 0;
    always #5 clk = ~clk;

    integer total_req;
    integer hit_count;
    integer total_cycles;
    integer start_time;
```

```
always @(posedge clk) begin
    if (start_time >= 0 && ready) begin
        total_cycles = total_cycles + (($time - start_time) / 10);
        start_time   = -1;
    end
end

always @(posedge clk) begin
    if (ready) begin
        total_req = total_req + 1;
        if (read && rdata !== 32'b0)
            hit_count = hit_count + 1;
    end
end

always @(posedge clk) begin
    $write("Time=%0t | State=", $time);
    case (fsm_state)
        uut.IDLE       : $write("IDLE       ");
        uut.READ_HIT   : $write("READ_HIT   ");
        uut.READ_MISS  : $write("READ_MISS  ");
        uut.WRITE_HIT  : $write("WRITE_HIT  ");
        uut.WRITE_MISS : $write("WRITE_MISS ");
        uut.EVICT      : $write("EVICT      ");
        default        : $write("UNKNOWN    ");
    endcase
    $write(" | Addr=0x%08h", addr);
    $write(" | Rdata=0x%08h", rdata);
    $write(" | Ready=%b", ready);
    $write("\n");
end

task do_read(input [31:0] a);
begin
    @(posedge clk);
    read       = 1;
```

```
    write      = 0;
    addr       = a;
    start_time = $time;
    @(posedge clk);
    read = 0;
    wait (ready);
    $display("[%-0t] READ(0x%08h) => 0x%08h", $time, a, rdata);
end
endtask

task do_write(input [31:0] a, input [31:0] d);
begin
    @(posedge clk);
    write      = 1;
    read       = 0;
    addr       = a;
    wdata      = d;
    start_time = $time;
    @(posedge clk);
    write = 0;
    wait (ready);
    $display("[%-0t] WRITE(0x%08h, 0x%08h)", $time, a, d);
end
endtask

initial begin
    reset       = 1;
    read        = 0;
    write       = 0;
    addr        = 32'b0;
    wdata       = 32'b0;
    total_req   = 0;
    hit_count   = 0;
    total_cycles= 0;
    start_time  = -1;
    #20;
```

```
        reset = 0;

        $display("\n=== STARTING CACHE CONTROLLER TEST ===\n");

        do_write(32'h0000_0010, 32'hDEAD_BEEF);

        do_read(32'h0000_0010);

        do_read(32'h0000_2000);

        do_write(32'h0000_2000, 32'h1234_5678);

        do_write(32'h0000_4000, 32'hAAAA_AAAA);
        do_write(32'h0000_6000, 32'hBBBB_BBBB);

        do_write(32'h0000_8000, 32'hCAFEBABE);

        $display("\n-- VERIFY EVICTION: READ from possibly evicted 0x0000_0000 --");
        do_read(32'h0000_0000);

        do_read(32'h0000_8000);

        do_write(32'h0000_0040, 32'h1111_1111);  // WRITE_MISS (tag=0, set=1)
        do_read (32'h0000_0040);                 // READ_HIT
        do_read (32'h0000_0840);                 // READ_MISS (tag=1, set=1)
        do_write(32'h0000_0840, 32'h2222_2222);  // WRITE_HIT

        #20;
        $display("\n=== TEST COMPLETE ===");
        $display("Total Requests : %0d", total_req);
        $display("Hit Count      : %0d", hit_count);
        $display("Hit Rate       : %0f", hit_count / (1.0 * total_req));
        $display("Avg. Latency   : %0f cycles", total_cycles / (1.0 * total_req));
        $finish;
    end
endmodule
```

Acesta contine urmatoarele componente esentiale:

- Generarea semnalului de ceas (clk) cu o perioada de 10ns.
- Initializarea semnalelor de control si adresare (reset, read, write, addr, wdata).
- Structura modulara ce include doua task-uri: `do_read` si `do_write`, fiecare simuland comportamentul procesorului pentru o operatie concreta.
- Monitorizarea starii interne a FSM-ului (`fsm_state`) si validarea valorii returnate (`rdata`), precum si activarea semnalului `ready`.

Fiecare task seteaza semnalele corespunzatoare, inregistreaza timpul de start si asteapta finalizarea operatiei. Valorile sunt apoi afisate folosind $display, oferind o vedere detaliata asupra comportamentului controlerului la nivel de ciclu de ceas.

Testele simulate includ:

- Scrierea initiala la o adresa si citirea ulterioara (WRITE → READ_HIT)
- Citirea unei adrese inexistente (READ_MISS)
- Inlocuirea blocului (EVICT)

Testbench-ul are rolul de a simula comportamentul unui procesor care interactioneaza cu cache-ul si de a verifica toate scenariile de acces posibile. Este construit modular, usor de urmarit si include logica pentru generarea de cereri, monitorizarea starilor si analiza performantelor.

Structura generala a testbench-ului include:

- ***Initializarea sistemului*** (reset = 1) pentru a porni controllerul intr-o stare curata.

- ***Crearea unui semnal de ceas stabil (clk)*** care oscileaza la fiecare 5 unitati de timp, mentinand sincronizarea tuturor proceselor.
- ***Declararea si resetarea variabilelor de contor***: total_req, hit_count, total_cycles, start_time, folosite pentru a masura performanta in timp real.

Comportamentul functional este construit pe baza a doua task-uri esentiale:

1. ***do_read(input a)*** – seteaza semnalele pentru o cerere de citire (read = 1), asteapta confirmarea (ready = 1) si afiseaza timpul de acces, adresa si datele citite.
2. ***do_write(input a, d)*** – seteaza semnalele pentru scriere (write = 1, wdata = d) si urmareste in mod similar finalizarea operatiei.

Monitorizarea FSM:

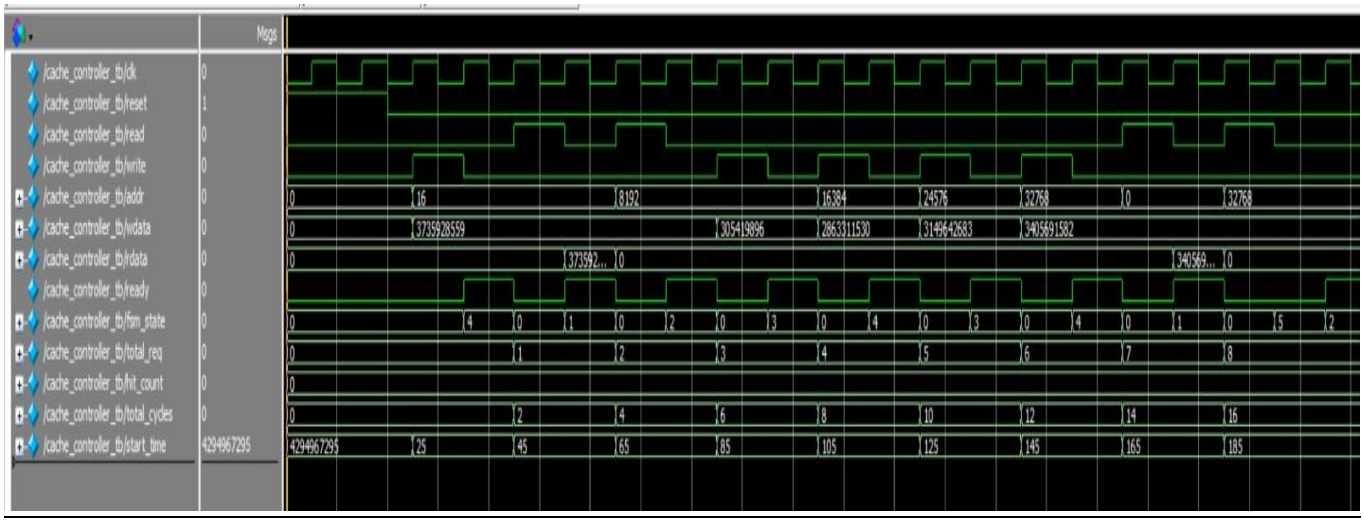
La fiecare ciclu de ceas, testbench-ul verifica starea actuala a controllerului prin semnalul fsm_state. Acesta poate avea valorile:

- ***IDLE*** – asteapta o comanda
- ***READ_HIT / WRITE_HIT*** – operatie completata direct din cache
- ***READ_MISS / WRITE_MISS*** – datele lipsesc, se aloca bloc nou
- ***EVICT*** – se elibereaza o cale prin politica LRU.

Schimbarea starilor este vizibila si sincronizata. Spre exemplu:

- Daca procesorul trimite o comanda read, iar hit = 1, FSM trece imediat in READ_HIT, semnalul ready este activat si se intoarce in IDLE.
- Daca hit = 0, dar exista o linie libera (free_found = 1), se aloca un bloc nou si FSM trece in READ_MISS.
- In cazul in care nu este nici un loc liber, FSM merge in EVICT, apoi in READ_MISS sau WRITE_MISS, in functie de tipul operatiei memorate (pending_rw).

- **6. WAVE:**

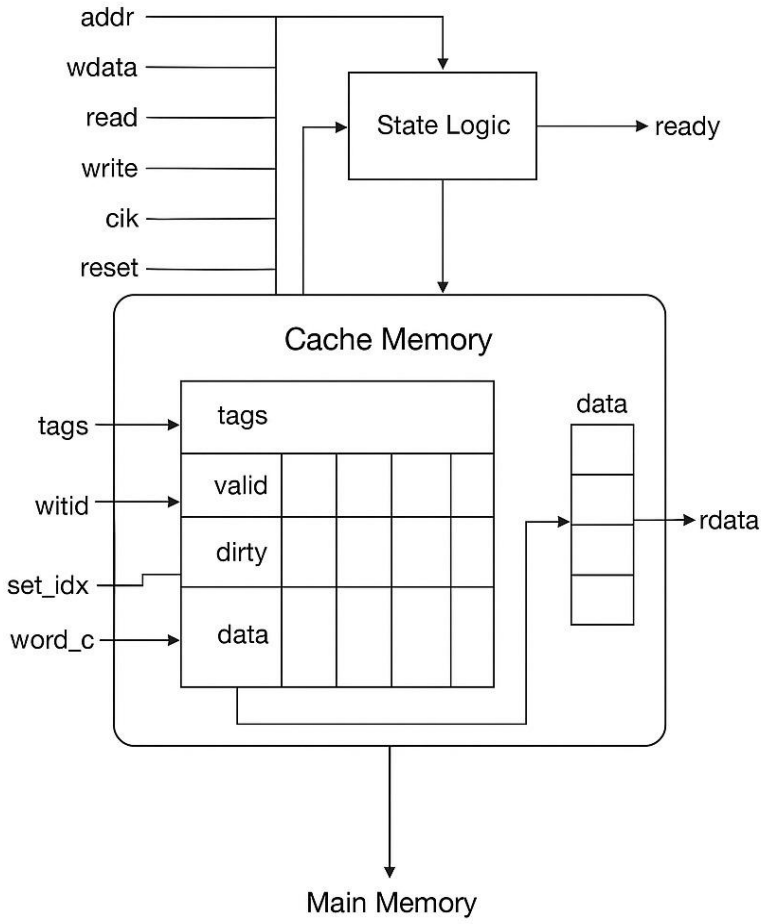


Toate comportamentele descrise in testbench se pot urmari clar in waveform-ul generat in timpul simularii.

In special, se pot observa urmatoarele:

- ***Tranzitia FSM-ului (fsm_state)***: schimbarile de stare sunt vizibile la fiecare operatie. De exemplu, o citire reusita va genera secventa IDLE → READ_HIT → IDLE, iar o scriere fara hit urmeaza traseul IDLE → WRITE_MISS → IDLE.
- ***Activarea semnalului ready***: apare exact in momentul cand controllerul a finalizat procesarea unei cereri, fie ca e HIT sau MISS. Se coreleaza cu iesirea din starea activa si intoarcerea in IDLE.
- ***Semnalele read si write***: se pot vedea ridicandu-se pentru un singur ciclu, la inceputul fiecarei operatii. Sunt urmate de modificari in addr, wdata, si de reactia FSM-ului.
- ***Timpul de raspuns (start_time pana la ready)***: permite masurarea latentei exacte pentru fiecare operatie. Acest timp este folosit la final pentru calculul *performantelor.*
- ***Datele returnate (rdata)****:* se modifica dupa un READ_HIT sau READ_MISS, si pot fi comparate cu ce a fost scris anterior prin wdata.

- # 7. DIAGRAMA BLOCK:



Aceasta diagrama ofera o vedere de ansamblu asupra structurii interne a controllerului de cache si a fluxului de date dintre procesor, cache si memoria principala.

Semnalele de intrare din partea procesorului:

- ***addr*** – adresa ceruta
- ***wdata*** – datele de scris
- ***read, write*** – semnale de control pentru tipul operatiei
- ***clk*** – semnalul de ceas (pentru sincronizare)

- ***reset*** – reseteaza sistemul

State Logic (FSM):

- Gestioneaza toate starile si tranzitiile in functie de operatia primita (READ/WRITE) si daca există HIT, MISS sau EVICT.
- Genereaza semnalul ready, indicand ca operatia este completa.

Cache Memory, contine mai multe componente:

- ***tags***: identifica ce adrese sunt stocate
- ***valid***: marcheaza liniile valide
- ***dirty***: indica daca o linie a fost modificata
- ***data*****:** memoreaza efectiv cuvintele stocate

->Daca datele sunt gasite (HIT), se returneaza rapid prin rdata.

->Daca nu sunt (MISS), datele sunt incarcate din memoria principala, prin conexiunea verticala din diagrama.

Alte intrari interne:

- ***set_idx***: indica setul selectat
- ***word_c***: offset-ul cuvantului in bloc
- ***witid***: identifica calea aleasa (way ID).

8. CONCLUZIE

Proiectul a urmarit implementarea practica a unui cache controller complet functional, integrand atat partea de logica secventiala (FSM), cat si gestiunea datelor cu structuri specifice memoriei cache.

Am reusit sa:

- Simulam si verificam toate cazurile de acces (hit, miss, evict)
- Utilizam concepte esentiale din arhitectura calculatoarelor precum valid/dirty bits, tag matching si LRU
- Implementam un design eficient si scalabil in Verilog
- Automatizam verificarea cu un testbench robust care masoara performanta controllerului

Rezultatul este un modul cache performant, cu latenta redusa si control logic clar, potrivit pentru a fi integrat intr-un sistem hardware mai complex.