



# **4-WAY SET ASSOCIATIVE CACHE SYSTEM (VERILOG)**

<b>Name</b>	<b>ID</b>
Ali Ahmed Mohamed Abdelmaboud	2200927
Hamza Basem Mohamed Ahmed	2200496
Amr Ahmed Wahidi	2200429

**SUBMITTED TO: DR. ASHRAF SLAEM  
DR. MOHAMED TAHER  
ENG. MOATSEM TAREK**

**DATE OF SUBMISSION: 25<sup>TH</sup> DEC. 2025**

## Introduction

This project presents the design and implementation of a simple memory hierarchy using Verilog HDL. The system includes a main memory (RAM), a 4-way set associative cache, and a cache controller. The cache reduces memory access latency by storing frequently accessed data closer to the processor.

---

## Objective

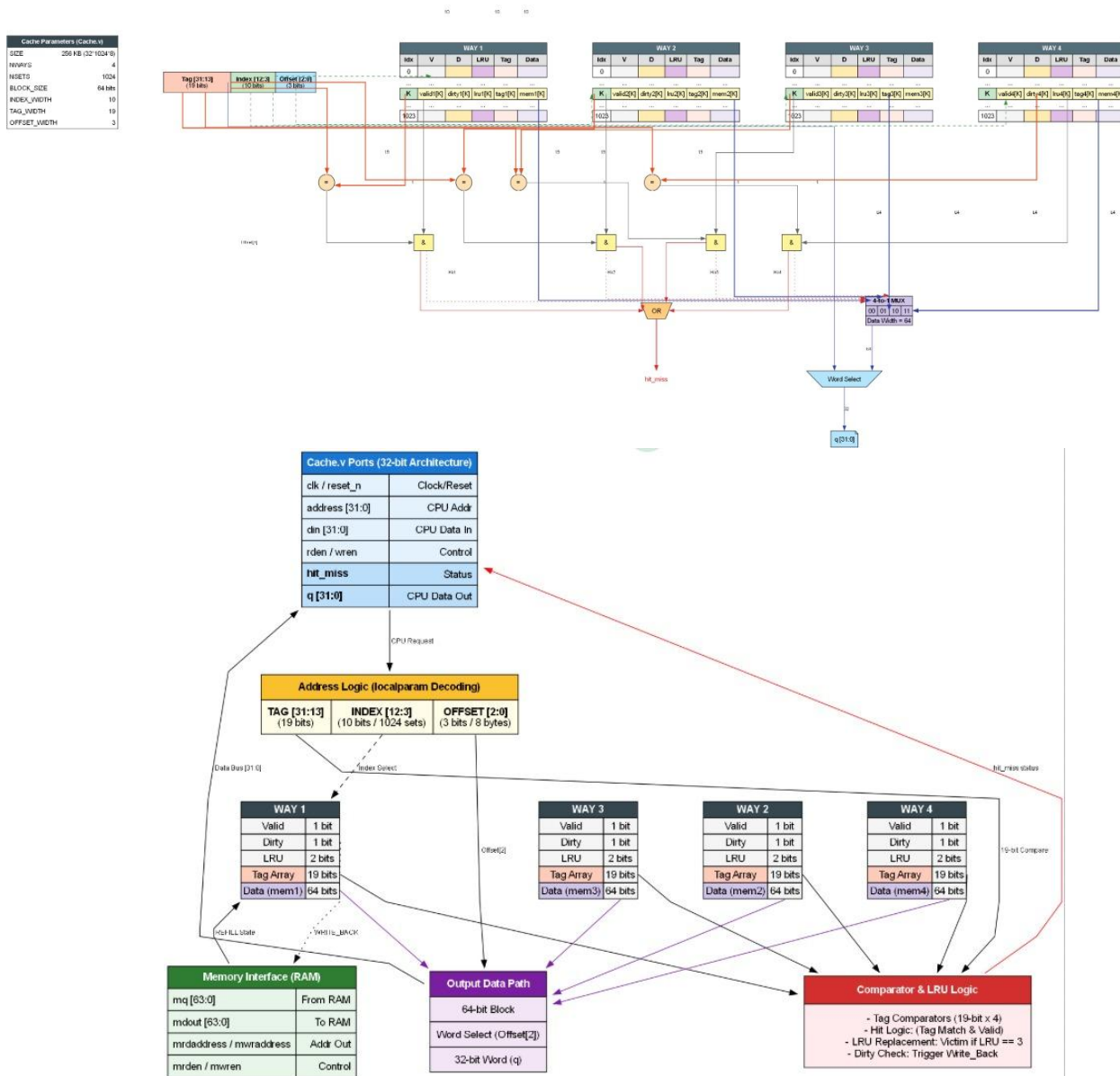
- Design a parameterized synchronous RAM
  - Implement a 4-way set associative cache
  - Implement cache controller logic for hit/miss handling
  - Apply LRU replacement policy (bonus)
  - Implement write-back using dirty bits (bonus)
- 

## System Overview

The CPU communicates with the cache first. On a cache hit, data is returned immediately. On a miss, the cache controller fetches data from main memory and updates the cache.

# CACHE SYSTEM ASSIGNMENT

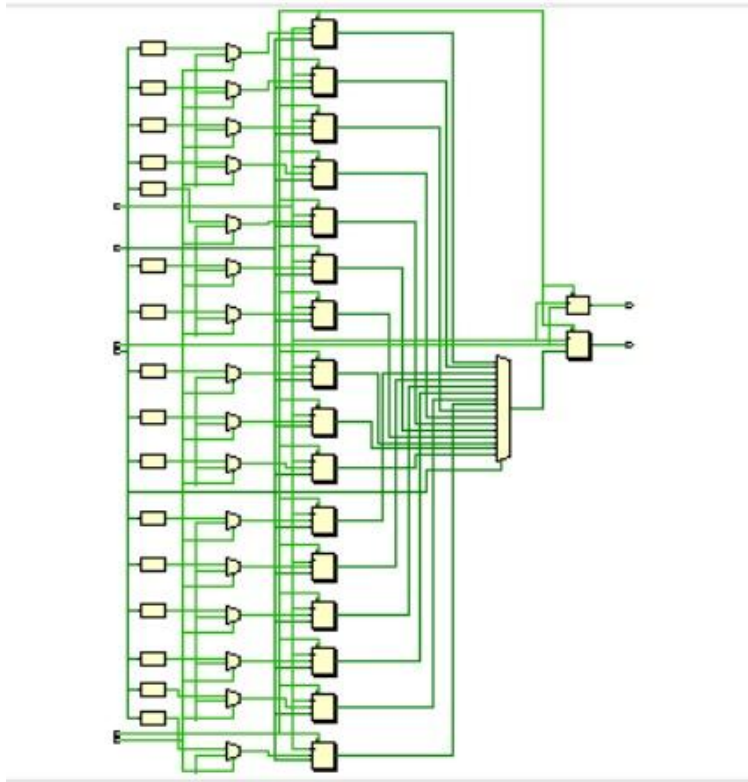
## Diagram



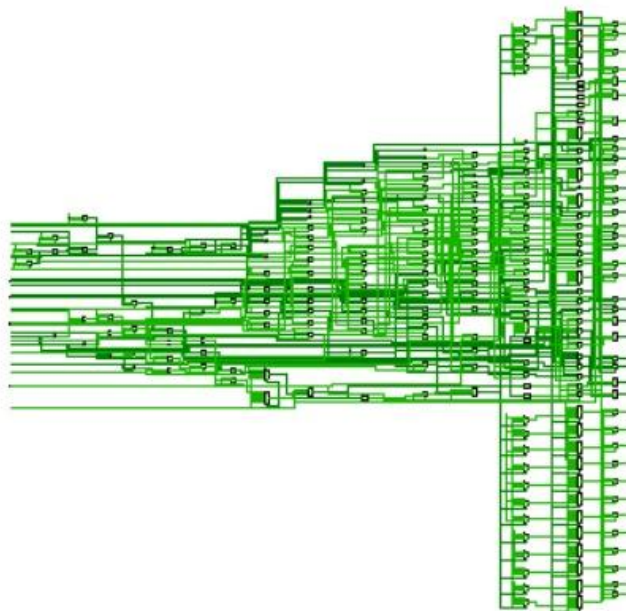
## Code of diagram

<https://github.com/amr10w/Caching System/tree/main/python%20code>

## Ram Module



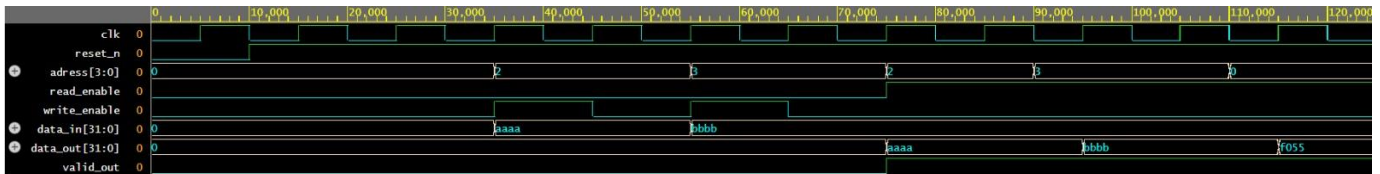
## Cach Module



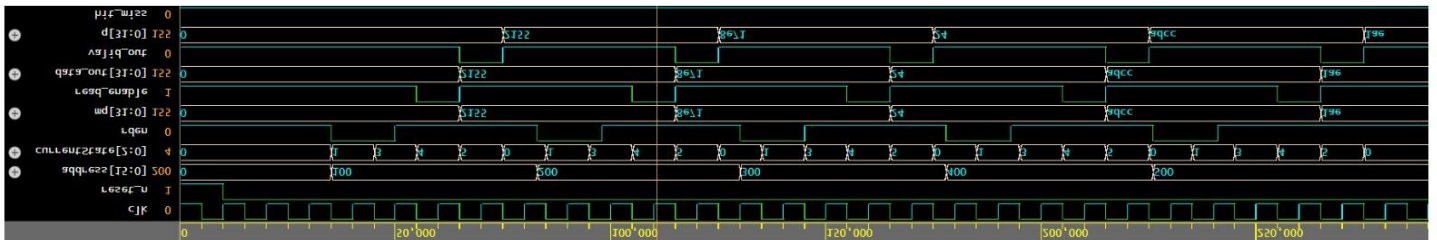
# CACHE SYSTEM ASSIGNMENT

## Waveforms

### Waveform Screenshot – Ram



### Waveform Screenshot – Cache Miss Handling

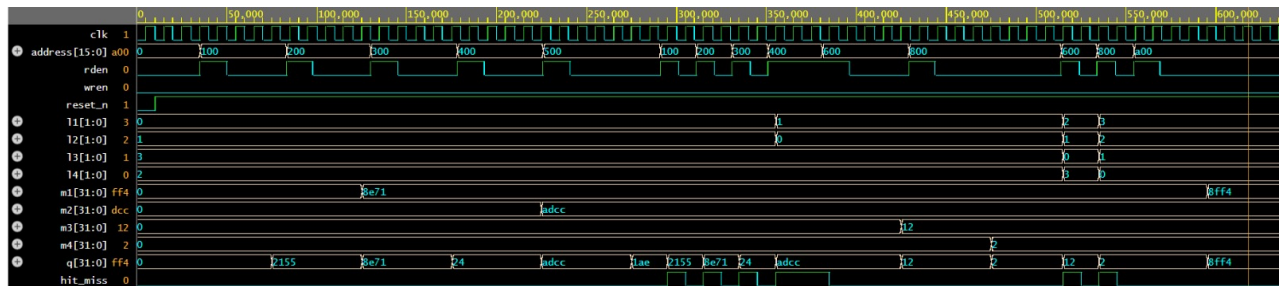
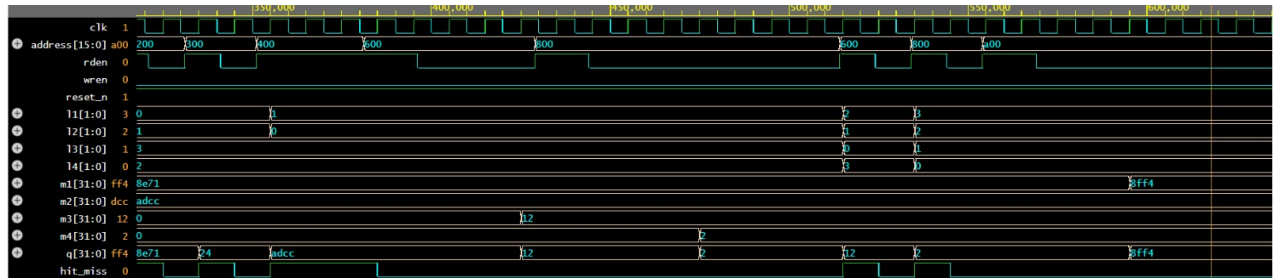
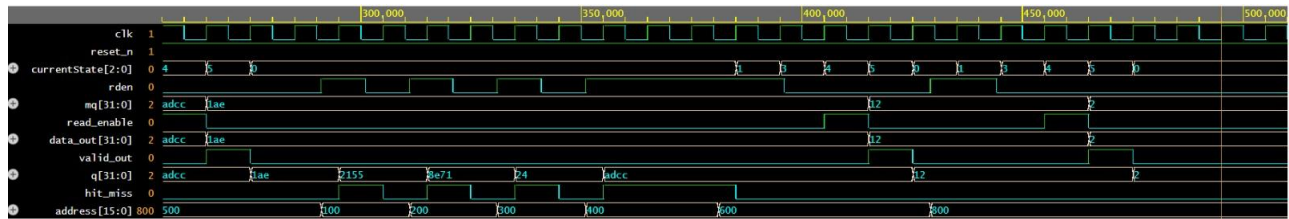


### Waveform Screenshot – Cache Hit

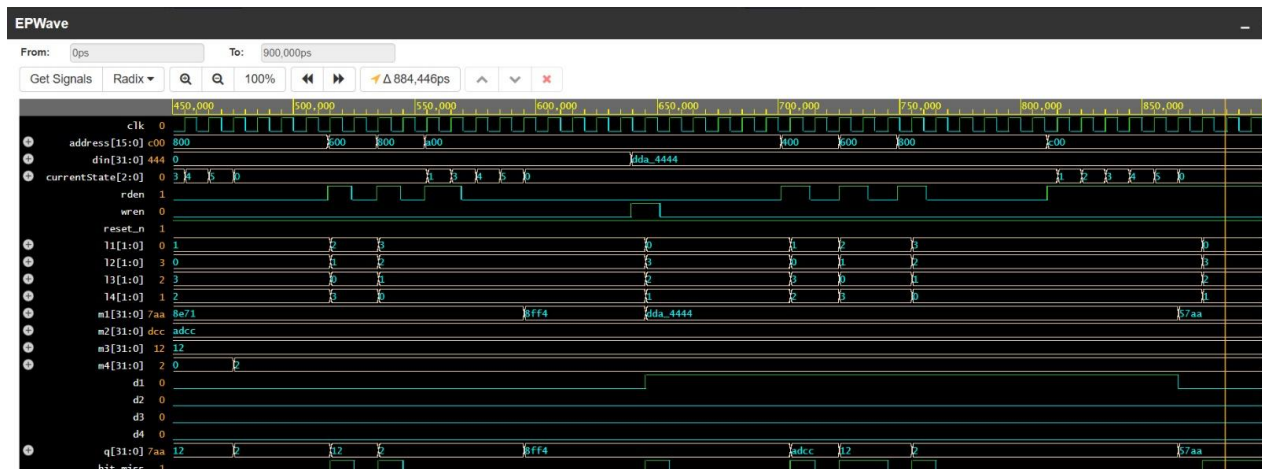


## CACHE SYSTEM ASSIGNMENT

## Waveform Screenshot – LRU Eviction



## Waveform Screenshot – Write-Back Operation



```
# Time: 0 | RAM[0x0a00] updated to: 00000000  
# Time: 25000 | RAM[0x0a00] updated to: 00008ff4  
# Time: 845000 | RAM[0x0a00] updated to: 0dda4444
```



## Verilog Source Code

### tb\_cache\_system.v

```
`timescale 1ns/1ps

module tb_cache_system;

    // Parameters
    parameter WIDTH = 32;
    parameter MWIDTH = 32; // Block size
    parameter ADDR_WIDTH = 16; // As per test requirement (approx)
        // Cache default is 32-bit address, but test uses 16.
        // We need to map 16 to Cache.v's expected width if necessary.
        // Cache.v defaults to WIDTH=32 for address?
        // Let's override Cache parameters.

    // Signals
    reg clk;
    reg reset_n;
    reg [ADDR_WIDTH-1:0] address;
    reg [WIDTH-1:0] din;
    reg wren;
    reg rden;

    wire [WIDTH-1:0] q;
    wire hit_miss;

    // RAM Interface Signals form Cache
    wire [MWIDTH-1:0] mdout;
    wire [ADDR_WIDTH-1:0] mrdaddress;
    wire      mrden;
    wire [ADDR_WIDTH-1:0] mwraddress;
    wire      mwren;
    wire [MWIDTH-1:0] mq; // Input to Cache (from RAM)

    // Signals for RAM Module
    wire [MWIDTH-1:0] ram_data_in;
    wire [ADDR_WIDTH-1:0] ram_address; // RAM depth is parameterized, likely 16 bits
    wire ram_write_enable;
    wire ram_read_enable;
    wire [MWIDTH-1:0] ram_data_out;
    wire ram_valid_out;

    // Cache Instance
    // Note: Cache default ADDR width is WIDTH=32. We can drive it with 32.
    // RAM we will make it 2^16 depth = 16 bit address.
    // We need to connect Cache (32b addr) to Ram (16b addr). Truncate.
```



```

Cache #(
    .WIDTH(WIDTH),
    .MWIDTH(MWIDTH),
    .NSETS(64),
    .NWAYS(4),
    .BLOCK_SIZE(MWIDTH), // Check logic
    .INDEX_WIDTH(6),
    .TAG_WIDTH(8),
    .OFFSET_WIDTH(3)

) dut_cache (
    .clk(clk),
    .reset_n(reset_n),
    .address(address),
    .din(din),
    .rden(rden),
    .wren(wren),
    .hit_miss(hit_miss),
    .q(q),

    .mdout(mdout),
    .mrdaddress(mrdaddress),
    .mrden(mrden),
    .mwraddress(mwraddress),
    .mwren(mwren),
    .mq(mq)
);

// RAM Interconnect Logic
// Cache has separate read/write ports. RAM has one.
// FSM ensures they are not active same time (Checked: WRITE_BACK then FETCH).
// So simple mux.

assign ram_write_enable = mwren;
assign ram_read_enable = mrden;
assign ram_address      = mwren ? mwraddress[ADDR_WIDTH-1:0] :
    mrdaddress[ADDR_WIDTH-1:0];
assign ram_data_in      = mdout;
assign mq                = ram_data_out;
wire [1:0] l1;
wire [1:0] l2;
wire [1:0] l3;
    wire [1:0] l4;
wire [31:0] m1;
wire [31:0] m2;
wire [31:0] m3;
wire [31:0] m4;
wire d1;
wire d2;
wire d3;

```

```

wire d4;
// RAM Module
Ram #(
    .WIDTH(MWIDTH), // RAM stores BLOCKS (64 bits)
    .DEPTH(ADDR_WIDTH) // 16 bits address
) dut_ram (
    .clk(clk),
    .reset_n(reset_n),
    .data_in(ram_data_in),
    .adress(ram_address),
    .write_enable(ram_write_enable),
    .read_enable(ram_read_enable),
    .data_out(ram_data_out),
    .valid_out(ram_valid_out)
);
    assign l1=dut_cache.lru1[0];
    assign l2=dut_cache.lru2[0];
    assign l3=dut_cache.lru3[0];
    assign l4=dut_cache.lru4[0];
    assign m1=dut_cache.mem1[0];
    assign m2=dut_cache.mem2[0];
    assign m3=dut_cache.mem3[0];
    assign m4=dut_cache.mem4[0];
    assign d1=dut_cache.dirty1[0];
assign d2=dut_cache.dirty2[0];
assign d3=dut_cache.dirty3[0];
assign d4=dut_cache.dirty4[0];
// Clock
always #5 clk = ~clk;
    initial begin
        clk = 0;
        address=0;
        reset_n=0;
        rden=0;
        wren=0;
        din=0;

        #10 reset_n=1;

        #15;
        $readmemh("Test1.mem",dut_ram.mem);
        #10
        address = 16'h0100;
        rden = 1;
        #15 rden=0;
        #33
        address = 16'h0200;
        rden = 1;
        #15 rden=0;

```

```

#32
address = 16'h0300;
  rden = 1;
#15 rden=0;
#33
address = 16'h0400;
  rden = 1;
#15 rden=0;
#33
address = 16'h0500;
  rden = 1;
#15 rden=0;
#50
// Hits:
address = 16'h0100;
  rden = 1;
#10 rden = 0;
#10

address = 16'h0200;
  rden = 1;

#10 rden = 0;
#10

address = 16'h0300;
  rden = 1;

#10 rden = 0;
#10

address = 16'h0400;
  rden = 1;
#30
address = 16'h0600;
  rden = 1;
#15 rden=0;
#33
address = 16'h0800;
  rden = 1;
#15 rden=0;
#70
address = 16'h0600;
  rden = 1;
  #10 rden = 0;
  #10

address = 16'h0800;
  rden = 1;

```

```

        #10 rden = 0;
        #10

address = 16'h0a00;
    rden = 1;
#15 rden=0;
#70;

address=16'h0a00;
din= 32'h 0dda_4444;
wren = 1;
#12 wren=0;
#50;
address = 16'h0400;
    rden = 1;

#12 rden = 0;
#12

address = 16'h0600;
    rden = 1;

#12 rden = 0;
#12

address = 16'h0800;
    rden = 1;
#12 rden = 0;
#50;
address = 16'h0c00;
    rden = 1;
#12 rden=1;
#80;

$display("Time: %0t | RAM Data at 0x0a00: %h", $time, dut_ram.mem[16'h0a00]);

        $finish;

end

initial begin
    $dumpfile("wave.vcd");
    $dumpvars;
end

endmodule

```

**tb\_ram.v**

```

module tb_ram;

    parameter WIDTH = 32;
    parameter DEPTH = 4; // 16 locations

    reg [WIDTH-1:0] data_in;
    reg [DEPTH-1:0] address;
    reg write_enable;
    reg read_enable;
    reg clk;
    reg reset_n;
    wire [WIDTH-1:0] data_out;
    wire valid_out;

    Ram #(
        .WIDTH(WIDTH),
        .DEPTH(DEPTH)
    ) uut (
        .data_in(data_in),
        .adress(address),
        .write_enable(write_enable),
        .read_enable(read_enable),
        .clk(clk),
        .reset_n(reset_n),
        .data_out(data_out),
        .valid_out(valid_out)
    );

    always #5 clk = ~clk;

    initial begin
        // Initialize
        clk = 0;
        reset_n = 0;
        write_enable = 0;
        read_enable = 0;
        data_in = 0;
        adress = 0;

        #10 reset_n=1;

        #15;
        $readmemh("Test1.mem",uut.mem);

        // Write 0xAAAA at address 2
        #10;

```

```

    adress = 4'd2;
    data_in = 32'hAAAA;
    write_enable = 1;
    #10;
    write_enable = 0;

    // Write 0xB BBB at address 3
    #10;
    adress = 4'd3;
    data_in = 32'hBBBB;
    write_enable = 1;
    #10;
    write_enable = 0;

    // IDLE
    #10;

    // Read address 2
    adress = 4'd2;
    read_enable = 1;

    #15;
    adress = 4'd3;

    #20;
    adress = 4'd0;
    #20;
    $finish;
End
initial begin
    $monitor("Time: %0t | RAM[0x0a00] updated to: %h",
        $time, dut_ram.mem[16'h0a00]);
end
initial begin
    $dumpfile("wave.vcd");
    $dumpvars;
end
endmodule

```

## design.v

```

`timescale 1ns/1ps

module Ram #(
    parameter WIDTH = 32,
    parameter DEPTH = 4
)(

```

```

input wire [WIDTH-1:0] data_in,
input wire [DEPTH-1:0] adress,
input wire write_enable,
input wire read_enable,
input wire clk,
input wire reset_n,
output reg [WIDTH-1:0] data_out,
output reg valid_out
);

localparam DEPTH_MEM = 1 << DEPTH;
reg [WIDTH-1:0] mem [0:DEPTH_MEM-1];

integer k;
always @(posedge clk or negedge reset_n) begin
    if (!reset_n) begin
        data_out <= 0;
        valid_out <= 1'b0;
        for (k = 0; k < DEPTH_MEM; k=k+1) begin
            mem[k] <= 0;
        end
    end
    else begin
        valid_out <= 1'b0;
        if (write_enable) begin
            mem[adress] <= data_in;
        end

        if (read_enable) begin
            data_out <= mem[adress];
            valid_out <= 1'b1;
        end
    end
end

```



```

    end
end

endmodule

module Cache
#(
    // Cache parameters
    parameter SIZE = 32*1024*8,
    parameter NWAYS = 4,
    parameter NSETS = 1024,
    parameter BLOCK_SIZE = 64,
    parameter WIDTH = 32,
    // Memory related parameter, make sure it matches memory module
    parameter MWIDTH = 64, // same as block size
    // More cache related parameters
    parameter INDEX_WIDTH = 10,
    parameter TAG_WIDTH = 19,
    parameter OFFSET_WIDTH = 3,
    parameter WORD1 = 3,
    parameter WORD2 = 7
)
(
    input wire      clk,      // renamed from clock
    input wire      reset_n,  // active low reset
    input wire [WIDTH-1:0]  address, // address from CPU
    input wire [WIDTH-1:0]  din,    // data from CPU (if st inst)
    input wire      rden,      // 1 if ld instruction
    input wire      wren,      // 1 if st instruction
    output wire      hit_miss, // 1 if hit, 0 while handling miss
    output wire [WIDTH-1:0]  q,     // data from cache to CPU

    // Memory Interface miss or write back

```

```

output wire [MWIDTH-1:0]    mdout,    // data from cache to memory (write back)
output wire [WIDTH-1:0]    mrdaddress, // memory read address
output wire                mrden,    // read enable, 1 if reading from memory (miss)
output wire [WIDTH-1:0]    mwraddress, // memory write address
output wire                mwren,    // write enable, 1 if writing to memory (write back)
input wire [MWIDTH-1:0]    mq        // data coming from memory (miss)
);

```

```

// Address Decoding Parameters
localparam OFFSET_HIGH = OFFSET_WIDTH - 1;
localparam OFFSET_LOW = 0;
localparam INDEX_HIGH = INDEX_WIDTH + OFFSET_WIDTH - 1;
localparam INDEX_LOW = OFFSET_WIDTH;
localparam TAG_HIGH = WIDTH - 1;
localparam TAG_LOW = INDEX_WIDTH + OFFSET_WIDTH;

/*****
* Global Parameters and Initializations
*****/

```

```

// WAY 1 cache data
reg        valid1 [0:NSETS-1];
reg        dirty1 [0:NSETS-1];
reg [1:0]   lru1   [0:NSETS-1];
reg [TAG_WIDTH-1:0] tag1 [0:NSETS-1];
reg [MWIDTH-1:0] mem1 [0:NSETS-1] /* synthesis ramstyle = "M20K" */;

```

```

// WAY 2 cache data
reg        valid2 [0:NSETS-1];
reg        dirty2 [0:NSETS-1];
reg [1:0]   lru2   [0:NSETS-1];
reg [TAG_WIDTH-1:0] tag2 [0:NSETS-1];
reg [MWIDTH-1:0] mem2 [0:NSETS-1] /* synthesis ramstyle = "M20K" */;

```

```
// WAY 3 cache data
reg      valid3 [0:NSETS-1];
reg      dirty3 [0:NSETS-1];
reg [1:0]   lru3  [0:NSETS-1];
reg [TAG_WIDTH-1:0] tag3  [0:NSETS-1];
reg [MWIDTH-1:0] mem3  [0:NSETS-1] /* synthesis ramstyle = "M20K" */;

// WAY 4 cache data
reg      valid4 [0:NSETS-1];
reg      dirty4 [0:NSETS-1];
reg [1:0]   lru4  [0:NSETS-1];
reg [TAG_WIDTH-1:0] tag4  [0:NSETS-1];
reg [MWIDTH-1:0] mem4  [0:NSETS-1] /* synthesis ramstyle = "M20K" */;

// internal registers
reg      _hit_miss = 1'b0;
reg [WIDTH-1:0] _q = {WIDTH{1'b0}};
reg [MWIDTH-1:0] _mdout = {MWIDTH{1'b0}};
reg [WIDTH-1:0] _mwraddress = {WIDTH{1'b0}};
reg [WIDTH-1:0] _mrdaddress = {WIDTH{1'b0}};
reg      _mwren = 1'b0;
reg      _mrden = 1'b0;
reg [MWIDTH-1:0] new_block;

// output assignments of internal registers
assign hit_miss = _hit_miss;
assign mwren = _mwren;
assign mdout = _mdout;
assign mwraddress = _mwraddress;
assign mrden = _mrden;
assign mrdaddress = _mrdaddress;
```

```

assign q = _q;

// state parameters
localparam IDLE    = 3'b000;
localparam MISS    = 3'b001; // Processing miss (checking victim)
localparam WRITE_BACK = 3'b010; // Writing dirty line to memory
localparam FETCH    = 3'b011; // Fetching new line from memory (sending read)
localparam FETCH_WAIT = 3'b100; // Wait for RAM latency
localparam REFILL    = 3'b101; // Capturing memory data and updating cache

// state register
reg [2:0] currentState = IDLE;

// Helper variables for FSM
reg [1:0] victim_way;

/*****
* State Machine
*****/
integer k;
always @(posedge clk or negedge reset_n)
begin
    if (!reset_n) begin
        currentState <= IDLE;
        _mwren <= 0;
        _mrden <= 0;

        _hit_miss <= 0;

        for(k = 0; k < NSETS; k = k + 1)
            begin
                valid1[k] = 0;
                valid2[k] = 0;
                valid3[k] = 0;
            end
    end
end

```

```

    valid4[k] = 0;
    dirty1[k] = 0;
    dirty2[k] = 0;
    dirty3[k] = 0;
    dirty4[k] = 0;
    lru1[k] = 2'b00;
    lru2[k] = 2'b01;
    lru3[k] = 2'b11;
    lru4[k] = 2'b10;
    tag1[k]=0;
    tag2[k]=0;
    tag3[k]=0;
    tag4[k]=0;
    mem1[k] =0;
    mem2[k] =0;
    mem3[k] =0;
    mem4[k] =0;
    end
end
else begin
    case (currentState)
        IDLE: begin
            _mwren <= 0;
            _mrden <= 0;

            // Do nothing if no request
            if (!rden && !wren) begin
                _hit_miss <= 0;
                currentState<=IDLE;
            end

            // Check Hit
            else if (valid1[address[INDEX_HIGH:INDEX_LOW]] &&
                (tag1[address[INDEX_HIGH:INDEX_LOW]] == address[TAG_HIGH:TAG_LOW])) begin

```

```

// ---- WAY 1 HIT ----
_hit_miss <= 1;
if (rden) begin
    _q <= (address[OFFSET_HIGH:OFFSET_LOW] <= WORD1) ?
mem1[address[INDEX_HIGH:INDEX_LOW]][WIDTH-1:0] :
mem1[address[INDEX_HIGH:INDEX_LOW]][2*WIDTH-1:WIDTH];
    end else if (wren) begin
        dirty1[address[INDEX_HIGH:INDEX_LOW]] <= 1;
        if (address[OFFSET_HIGH:OFFSET_LOW] <= WORD1)
mem1[address[INDEX_HIGH:INDEX_LOW]][WIDTH-1:0] <= din;
        else mem1[address[INDEX_HIGH:INDEX_LOW]][2*WIDTH-1:WIDTH] <= din;
    end
    // Update LRU
    if (lru2[address[INDEX_HIGH:INDEX_LOW]] <=
lru1[address[INDEX_HIGH:INDEX_LOW]]) lru2[address[INDEX_HIGH:INDEX_LOW]] <=
lru2[address[INDEX_HIGH:INDEX_LOW]] + 1;
    if (lru3[address[INDEX_HIGH:INDEX_LOW]] <=
lru1[address[INDEX_HIGH:INDEX_LOW]]) lru3[address[INDEX_HIGH:INDEX_LOW]] <=
lru3[address[INDEX_HIGH:INDEX_LOW]] + 1;
    if (lru4[address[INDEX_HIGH:INDEX_LOW]] <=
lru1[address[INDEX_HIGH:INDEX_LOW]]) lru4[address[INDEX_HIGH:INDEX_LOW]] <=
lru4[address[INDEX_HIGH:INDEX_LOW]] + 1;
    lru1[address[INDEX_HIGH:INDEX_LOW]] <= 0;
end
    else if (valid2[address[INDEX_HIGH:INDEX_LOW]] &&
(tag2[address[INDEX_HIGH:INDEX_LOW]] == address[TAG_HIGH:TAG_LOW])) begin
        // ---- WAY 2 HIT ----
        _hit_miss <= 1;
        if (rden) begin
            _q <= (address[OFFSET_HIGH:OFFSET_LOW] <= WORD1) ?
mem2[address[INDEX_HIGH:INDEX_LOW]][WIDTH-1:0] :
mem2[address[INDEX_HIGH:INDEX_LOW]][2*WIDTH-1:WIDTH];
            end else if (wren) begin
                dirty2[address[INDEX_HIGH:INDEX_LOW]] <= 1;
                if (address[OFFSET_HIGH:OFFSET_LOW] <= WORD1)
mem2[address[INDEX_HIGH:INDEX_LOW]][WIDTH-1:0] <= din;
                else mem2[address[INDEX_HIGH:INDEX_LOW]][2*WIDTH-1:WIDTH] <= din;
            end
        end
    end
end

```

```

// Update LRU
    if (lru1[address[INDEX_HIGH:INDEX_LOW]] <=
lru2[address[INDEX_HIGH:INDEX_LOW]]) lru1[address[INDEX_HIGH:INDEX_LOW]] <=
lru1[address[INDEX_HIGH:INDEX_LOW]] + 1;

    if (lru3[address[INDEX_HIGH:INDEX_LOW]] <=
lru2[address[INDEX_HIGH:INDEX_LOW]]) lru3[address[INDEX_HIGH:INDEX_LOW]] <=
lru3[address[INDEX_HIGH:INDEX_LOW]] + 1;

    if (lru4[address[INDEX_HIGH:INDEX_LOW]] <=
lru2[address[INDEX_HIGH:INDEX_LOW]]) lru4[address[INDEX_HIGH:INDEX_LOW]] <=
lru4[address[INDEX_HIGH:INDEX_LOW]] + 1;

    lru2[address[INDEX_HIGH:INDEX_LOW]] <= 0;
end

else if (valid3[address[INDEX_HIGH:INDEX_LOW]] &&
(tag3[address[INDEX_HIGH:INDEX_LOW]] == address[TAG_HIGH:TAG_LOW])) begin
    // ---- WAY 3 HIT ----
    _hit_miss <= 1;
    if (rden) begin
        _q <= (address[OFFSET_HIGH:OFFSET_LOW] <= WORD1) ?
mem3[address[INDEX_HIGH:INDEX_LOW]][WIDTH-1:0] :
mem3[address[INDEX_HIGH:INDEX_LOW]][2*WIDTH-1:WIDTH];
    end else if (wren) begin
        dirty3[address[INDEX_HIGH:INDEX_LOW]] <= 1;
        if (address[OFFSET_HIGH:OFFSET_LOW] <= WORD1)
mem3[address[INDEX_HIGH:INDEX_LOW]][WIDTH-1:0] <= din;
        else mem3[address[INDEX_HIGH:INDEX_LOW]][2*WIDTH-1:WIDTH] <= din;
    end
    // Update LRU
    if (lru1[address[INDEX_HIGH:INDEX_LOW]] <=
lru3[address[INDEX_HIGH:INDEX_LOW]]) lru1[address[INDEX_HIGH:INDEX_LOW]] <=
lru1[address[INDEX_HIGH:INDEX_LOW]] + 1;

    if (lru2[address[INDEX_HIGH:INDEX_LOW]] <=
lru3[address[INDEX_HIGH:INDEX_LOW]]) lru2[address[INDEX_HIGH:INDEX_LOW]] <=
lru2[address[INDEX_HIGH:INDEX_LOW]] + 1;

    if (lru4[address[INDEX_HIGH:INDEX_LOW]] <=
lru3[address[INDEX_HIGH:INDEX_LOW]]) lru4[address[INDEX_HIGH:INDEX_LOW]] <=
lru4[address[INDEX_HIGH:INDEX_LOW]] + 1;

    lru3[address[INDEX_HIGH:INDEX_LOW]] <= 0;
end

else if (valid4[address[INDEX_HIGH:INDEX_LOW]] &&
(tag4[address[INDEX_HIGH:INDEX_LOW]] == address[TAG_HIGH:TAG_LOW])) begin

```



```

// ---- WAY 4 HIT ----
_hit_miss <= 1;
if (rden) begin
    _q <= (address[OFFSET_HIGH:OFFSET_LOW] <= WORD1) ?
mem4[address[INDEX_HIGH:INDEX_LOW]][WIDTH-1:0] :
mem4[address[INDEX_HIGH:INDEX_LOW]][2*WIDTH-1:WIDTH];
    end else if (wren) begin
        dirty4[address[INDEX_HIGH:INDEX_LOW]] <= 1;
        if (address[OFFSET_HIGH:OFFSET_LOW] <= WORD1)
mem4[address[INDEX_HIGH:INDEX_LOW]][WIDTH-1:0] <= din;
        else mem4[address[INDEX_HIGH:INDEX_LOW]][2*WIDTH-1:WIDTH] <= din;
    end
    // Update LRU
    if (lru1[address[INDEX_HIGH:INDEX_LOW]] <=
lru4[address[INDEX_HIGH:INDEX_LOW]]) lru1[address[INDEX_HIGH:INDEX_LOW]] <=
lru1[address[INDEX_HIGH:INDEX_LOW]] + 1;
    if (lru2[address[INDEX_HIGH:INDEX_LOW]] <=
lru4[address[INDEX_HIGH:INDEX_LOW]]) lru2[address[INDEX_HIGH:INDEX_LOW]] <=
lru2[address[INDEX_HIGH:INDEX_LOW]] + 1;
    if (lru3[address[INDEX_HIGH:INDEX_LOW]] <=
lru4[address[INDEX_HIGH:INDEX_LOW]]) lru3[address[INDEX_HIGH:INDEX_LOW]] <=
lru3[address[INDEX_HIGH:INDEX_LOW]] + 1;
    lru4[address[INDEX_HIGH:INDEX_LOW]] <= 0;
end
else begin
    // ---- MISS ----
    _hit_miss <= 0;
    currentState <= MISS; // next postive_edge/state we will handle the miss
end
end

MISS: begin
    // Check if any way is invalid (Empty)
    if (!valid1[address[INDEX_HIGH:INDEX_LOW]]) begin
        victim_way <= 2'b00;
        currentState <= FETCH;
    end
end

```

```

else if (!valid2[address[INDEX_HIGH:INDEX_LOW]]) begin
    victim_way <= 2'b01;
    currentState <= FETCH;
end
else if (!valid3[address[INDEX_HIGH:INDEX_LOW]]) begin
    victim_way <= 2'b10;
    currentState <= FETCH;
end
else if (!valid4[address[INDEX_HIGH:INDEX_LOW]]) begin
    victim_way <= 2'b11;
    currentState <= FETCH;
end
// If all valid, Check LRU and Dirty Status
else begin
    // LRU is Way 1
    if (lru1[address[INDEX_HIGH:INDEX_LOW]] == 3) begin
        victim_way <= 2'b00;
        if (dirty1[address[INDEX_HIGH:INDEX_LOW]]) begin
            currentState <= WRITE_BACK;
        end else currentState <= FETCH;
    end
    // LRU is Way 2
    else if (lru2[address[INDEX_HIGH:INDEX_LOW]] == 3) begin
        victim_way <= 2'b01;
        if (dirty2[address[INDEX_HIGH:INDEX_LOW]]) begin
            currentState <= WRITE_BACK;
        end else currentState <= FETCH;
    end
    // LRU is Way 3
    else if (lru3[address[INDEX_HIGH:INDEX_LOW]] == 3) begin
        victim_way <= 2'b10;
        if (dirty3[address[INDEX_HIGH:INDEX_LOW]]) begin
            currentState <= WRITE_BACK;
        end else currentState <= FETCH;
    end
end

```

```

end
// LRU is Way 4
else begin
    victim_way <= 2'b11;
    if (dirty4[address[INDEX_HIGH:INDEX_LOW]]) begin
        currentState <= WRITE_BACK;
    end else currentState <= FETCH;
end
end
end

WRITE_BACK: begin
    _mwren <= 1;

    case (victim_way)
        2'b00: begin
            _mdout <= mem1[address[INDEX_HIGH:INDEX_LOW]];
            _mwraddress <= {tag1[address[INDEX_HIGH:INDEX_LOW]],
address[INDEX_HIGH:INDEX_LOW], {OFFSET_WIDTH{1'b0}}};
        end
        2'b01: begin
            _mdout <= mem2[address[INDEX_HIGH:INDEX_LOW]];
            _mwraddress <= {tag2[address[INDEX_HIGH:INDEX_LOW]],
address[INDEX_HIGH:INDEX_LOW], {OFFSET_WIDTH{1'b0}}};
        end
        2'b10: begin
            _mdout <= mem3[address[INDEX_HIGH:INDEX_LOW]];
            _mwraddress <= {tag3[address[INDEX_HIGH:INDEX_LOW]],
address[INDEX_HIGH:INDEX_LOW], {OFFSET_WIDTH{1'b0}}};
        end
        2'b11: begin
            _mdout <= mem4[address[INDEX_HIGH:INDEX_LOW]];
            _mwraddress <= {tag4[address[INDEX_HIGH:INDEX_LOW]],
address[INDEX_HIGH:INDEX_LOW], {OFFSET_WIDTH{1'b0}}};
        end
    end
end

```

```

endcase

currentState <= FETCH;
end

FETCH: begin
    _mwren <= 0;
    _mrden <= 1;
    _mraddress <= {address[TAG_HIGH:TAG_LOW],
address[INDEX_HIGH:INDEX_LOW], {OFFSET_WIDTH{1'b0}}};
    currentState <= FETCH_WAIT;
end

FETCH_WAIT: begin
    _mrden <= 0; // Deassert read enable
    currentState <= REFILL;
end

REFILL: begin
    _mrden <= 0;

    new_block = mq; // From memory

    if (wren) begin
        if (address[OFFSET_HIGH:OFFSET_LOW] <= WORD1) new_block[WIDTH-1:0] =
din;

        else new_block[2*WIDTH-1:WIDTH] = din;
    end

    case (victim_way)
        2'b00: begin // Way 1
            mem1[address[INDEX_HIGH:INDEX_LOW]] <= new_block;
            tag1[address[INDEX_HIGH:INDEX_LOW]] <= address[TAG_HIGH:TAG_LOW];
            valid1[address[INDEX_HIGH:INDEX_LOW]] <= 1;
            dirty1[address[INDEX_HIGH:INDEX_LOW]] <= wren;

```

```

end
2'b01: begin // Way 2
    mem2[address[INDEX_HIGH:INDEX_LOW]] <= new_block;
    tag2[address[INDEX_HIGH:INDEX_LOW]] <= address[TAG_HIGH:TAG_LOW];
    valid2[address[INDEX_HIGH:INDEX_LOW]] <= 1;
    dirty2[address[INDEX_HIGH:INDEX_LOW]] <= wren;
end
2'b10: begin // Way 3
    mem3[address[INDEX_HIGH:INDEX_LOW]] <= new_block;
    tag3[address[INDEX_HIGH:INDEX_LOW]] <= address[TAG_HIGH:TAG_LOW];
    valid3[address[INDEX_HIGH:INDEX_LOW]] <= 1;
    dirty3[address[INDEX_HIGH:INDEX_LOW]] <= wren;
end
2'b11: begin // Way 4
    mem4[address[INDEX_HIGH:INDEX_LOW]] <= new_block;
    tag4[address[INDEX_HIGH:INDEX_LOW]] <= address[TAG_HIGH:TAG_LOW];
    valid4[address[INDEX_HIGH:INDEX_LOW]] <= 1;
    dirty4[address[INDEX_HIGH:INDEX_LOW]] <= wren;
end
endcase
_q<=new_block;
currentState <= IDLE;
end
endcase
end
end
endmodule

```

## Conclusion

The implemented caching system successfully demonstrates cache hit/miss behavior, LRU replacement, and write-back consistency. Simulation results validate the correctness of the design.

---

## Repo & Video links

- Repo: <https://github.com/amr10w/Caching System>
- Video: <https://youtu.be/Mhmy6muTbB4>