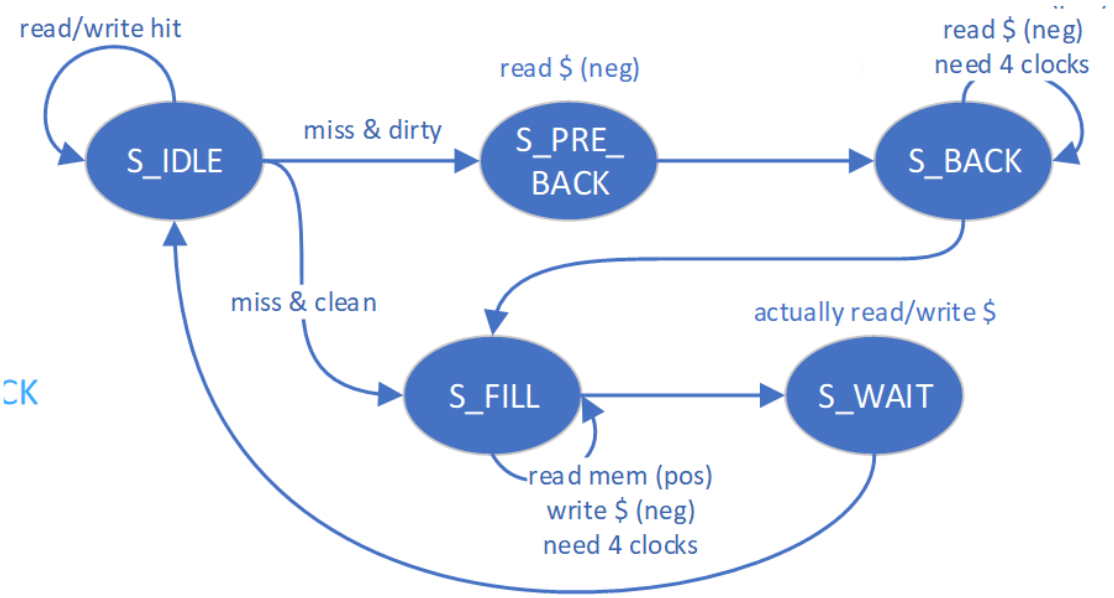
Lab4: Pipelined CPU with Cache

3220106039 李瀚轩

1. Experimental Steps

1.1 Compelete the CMU Module

According to the PPT, the key to the CMU module is completing the state machine setup, based on the diagram below:



After this, we can complete the framework code:

```
module cmu (
        // CPU side
        input clk,
        input rst,
        input [31:0] addr_rw,
        input en_r,
        input en_w,
        input [2:0] u_b_h_w,
        input [31:0] data_w,
        output [31:0] data_r,
        output stall,

        // mem side
        output reg mem_cs_o = 0,
        output reg mem_we_o = 0,
        output reg [31:0] mem_addr_o = 0,
        input [31:0] mem_data_i,
        output [31:0] mem_data_o,
        input mem_ack_i,

        // debug info
```

```
        output [2:0] cmu_state
    );

    `include "addr_define.vh"

    reg [ADDR_BITS-1:0] cache_addr = 0;
    reg cache_load = 0;
    reg cache_store = 0;
    reg cache_edit = 0;
    reg [2:0] cache_u_b_h_w = 0;
    reg [WORD_BITS-1:0] cache_din = 0;
    wire cache_hit;
    wire [WORD_BITS-1:0] cache_dout;
    wire cache_valid;
    wire cache_dirty;
    wire [TAG_BITS-1:0] cache_tag;

    cache CACHE (
        .clk(~clk),
        .rst(rst),
        .addr(cache_addr),
        .load(cache_load),
        .store(cache_store),
        .edit(cache_edit),
        .invalid(1'b0),
        .u_b_h_w(cache_u_b_h_w),
        .din(cache_din),
        .hit(cache_hit),
        .dout(cache_dout),
        .valid(cache_valid),
        .dirty(cache_dirty),
        .tag(cache_tag)
    );

    localparam
        S_IDLE = 0,
        S_PRE_BACK = 1,
        S_BACK = 2,
        S_FILL = 3,
        S_WAIT = 4;

    reg [2:0]state = 0;
    reg [2:0]next_state = 0;
    reg [ELEMENT_WORDS_WIDTH-1:0]word_count = 0;
    reg [ELEMENT_WORDS_WIDTH-1:0]next_word_count = 0;
    assign cmu_state = state;

    always @ (posedge clk) begin
        if (rst) begin
            state <= S_IDLE;
            word_count <= 2'b00;
        end
        else begin
            state <= next_state;
```

```
            word_count <= next_word_count;
        end
    end

    // state ctrl
    always @ (*) begin
        if (rst) begin
            next_state = S_IDLE;
            next_word_count = 2'b00;
        end
        else begin
            case (state)
                S_IDLE: begin
                    if (en_r || en_w) begin
                        if (cache_hit)
                            next_state = S_IDLE;
                        else if (cache_valid && cache_dirty)
                            next_state = S_PRE_BACK;
                        else
                            next_state = S_FILL;
                    end
                    next_word_count = 2'b00;
                end

                S_PRE_BACK: begin
                    next_state = S_BACK;
                    next_word_count = 2'b00;
                end

                S_BACK: begin
                    if (mem_ack_i && word_count ==
{ELEMENT_WORDS_WIDTH{1'b1}})    // 2'b11 in default case
                        next_state = S_FILL;
                    else
                        next_state = S_BACK;

                    if (mem_ack_i)
                        next_word_count = word_count + 2'b01;
                    else
                        next_word_count = word_count;
                end

                S_FILL: begin
                    if (mem_ack_i && word_count ==
{ELEMENT_WORDS_WIDTH{1'b1}})
                        next_state = S_WAIT;
                    else
                        next_state = S_FILL;

                    if (mem_ack_i)
                        next_word_count = word_count + 2'b01;
                    else
                        next_word_count = word_count;
                end
```

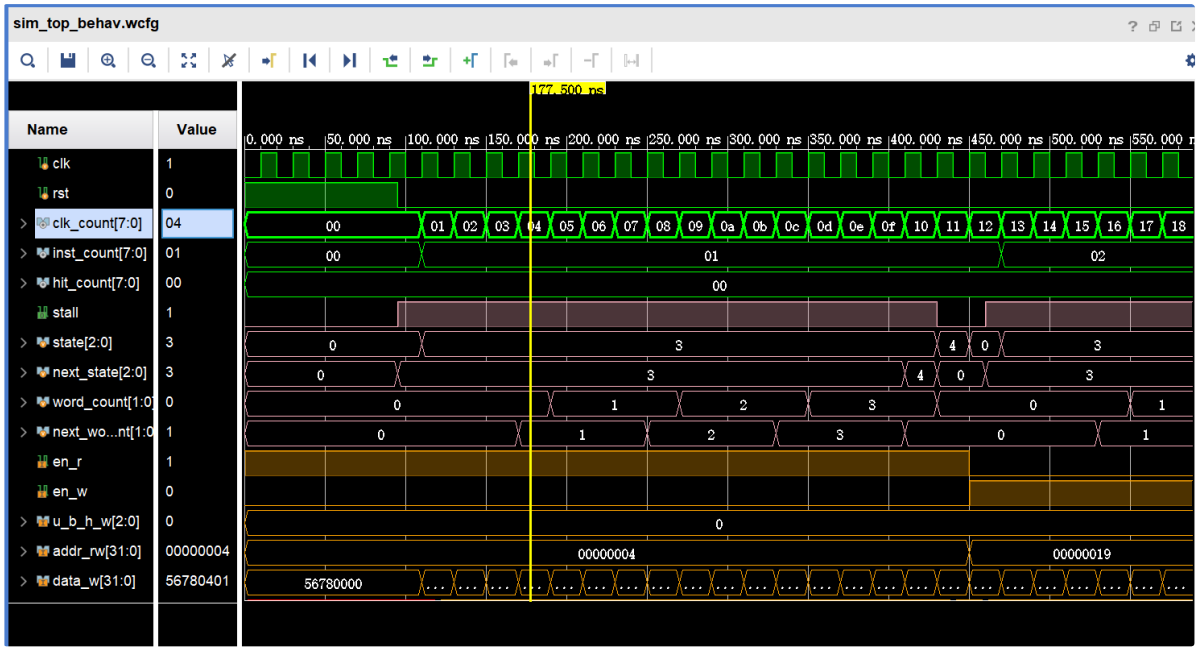
```
                S_WAIT: begin
                    next_state = S_IDLE;
                    next_word_count = 2'b00;
                end
            endcase
        end
    end

    // cache ctrl
    always @ (*) begin
        case(state)
            S_IDLE, S_WAIT: begin
                cache_addr = addr_rw;
                cache_load = en_r;
                cache_edit = en_w;
                cache_store = 1'b0;
                cache_u_b_h_w = u_b_h_w;
                cache_din = data_w;
            end
            S_BACK, S_PRE_BACK: begin
                cache_addr = {addr_rw[ADDR_BITS-1:BLOCK_WIDTH],
next_word_count, {ELEMENT_WORDS_WIDTH{1'b0}}};
                cache_load = 1'b0;
                cache_edit = 1'b0;
                cache_store = 1'b0;
                cache_u_b_h_w = 3'b010;
                cache_din = 32'b0;
            end
            S_FILL: begin
                cache_addr = {addr_rw[ADDR_BITS-1:BLOCK_WIDTH], word_count,
{ELEMENT_WORDS_WIDTH{1'b0}}};
                cache_load = 1'b0;
                cache_edit = 1'b0;
                cache_store = mem_ack_i;
                cache_u_b_h_w = 3'b010;
                cache_din = mem_data_i;
            end
        endcase
    end
    assign data_r = cache_dout;

    // mem ctrl
always @ (*) begin
    case (next_state)
        S_IDLE, S_PRE_BACK, S_WAIT: begin
            mem_cs_o = 1'b0;
            mem_we_o = 1'b0;
            mem_addr_o = 32'b0;
        end

        S_BACK: begin
            mem_cs_o = 1'b1;
            mem_we_o = 1'b1;
```

```
            mem_addr_o = {cache_tag, addr_rw[ADDR_BITS-TAG_BITS-
1:BLOCK_WIDTH], next_word_count, {ELEMENT_WORDS_WIDTH{1'b0}}};
        end

        S_FILL: begin
            mem_cs_o = 1'b1;
            mem_we_o = 1'b0;
            mem_addr_o = {addr_rw[ADDR_BITS-1:BLOCK_WIDTH], next_word_count,
{ELEMENT_WORDS_WIDTH{1'b0}}};
        end
    endcase
end
assign mem_data_o = cache_dout;

    assign stall = next_state != S_IDLE;

endmodule
```

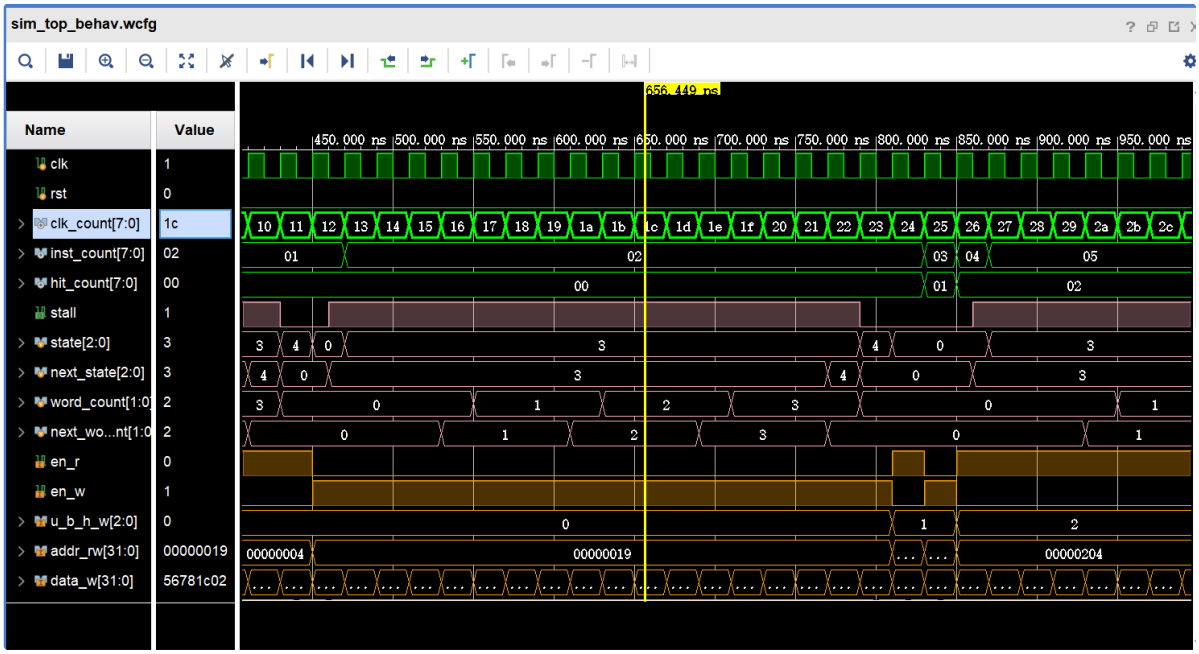
1.2 Completing the Cache Module

Here, we can simply bring over the code from lab3 without further explanation.

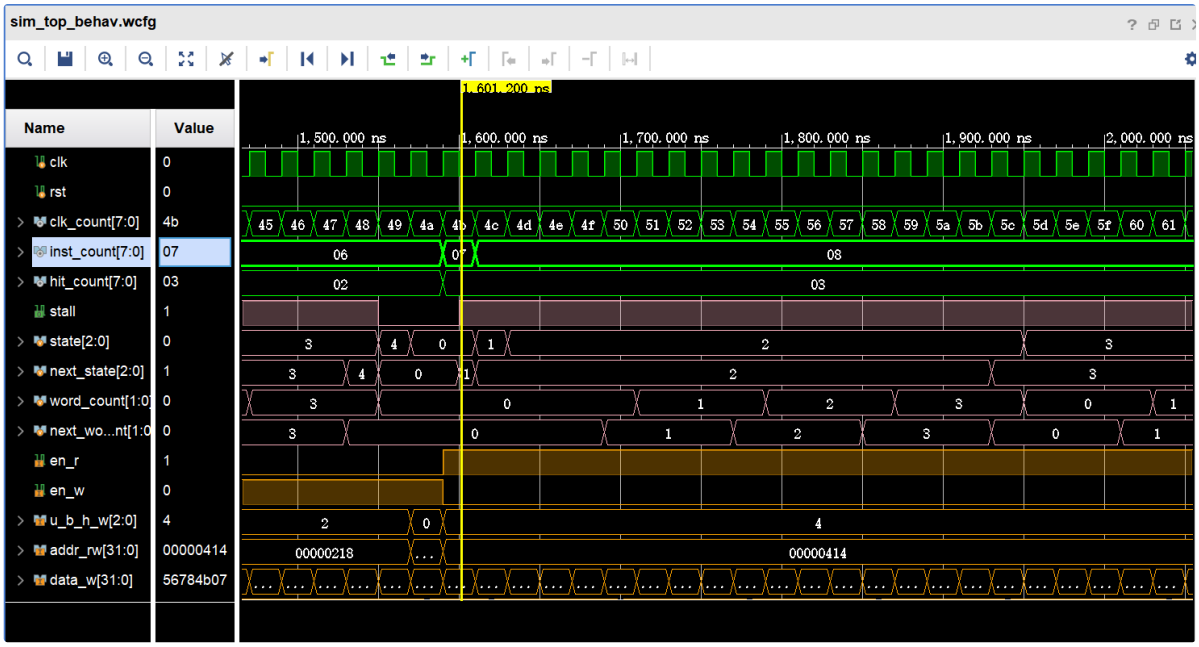
2. Evaluation



The instruction at `0x4` is a load instruction. Since the cache is empty, a read miss occurs. As a result, we can observe 0x8 and 0xC changing sequentially (with `mem_addr_o` going from 0x0 to 0x4 and `cache_store=1`), which loads the corresponding data into the cache. Here, you can see that it takes 4*4=16 cycles (with `clk_count` changing from 0 to 0x10).



For the third and fourth instructions, a cache hit occurs. At 830ns and 850ns, you can see that one is a read hit and the other is a write hit, each taking only one cycle.



For the eighth instruction, a miss occurs, and a write-back is required. At 1610ns, the state changes from 0 to 1 (indicating `S_PRE_BACK`), and in the next cycle, it changes to 2 (indicating `S_BACK`). During this phase, the cache data is written back, and then it moves into the `S_FILL` phase, each requiring 4*4=16 cycles.