浙江大学

本科实验报告

课程名称：	计算机体系结构
姓 名：	张志心
学 院：	竺可桢学院
专 业：	混合班
学 号：	3210106357
指导教师：	常瑞
日 期：	2023 年 12 月 14 日

浙江大学实验报告

课程名称： 计算机体系结构 实验类型： 综合

实验项目名称： 实验4 - L1 cache设计

学生姓名： 张志心 专业： 计算机科学与技术 学号： 3210106357

同组学生姓名： 无 指导教师： 常瑞 助教： 邱明冉

实验地点： 曹光彪西301 实验日期： 2023 年 12 月 14 日

1 实验目的

- 了解cache在CPU中的作用
- 了解cache management unit(CMU)与cache和memory之间的交互机制
- 在CPU中集成cache

2 实验环境

- **HDL**：Verilog、SystemVerilog
- **IDE**：Vivado
- **开发板**：NEXYS A7（XC7A100TCSG324）

3 实验原理

3.1 Cache

本次实验要求cache实现**2路组相联**，采用write-back,write-allocate的数据更新策略，采用LRU的替换策略。

cache block如图1所示。此外，本次实验实现的Cache大小为1024B，每个块有16B，因此合计有1024/16/2=32个set。

LRU	V	D	Tag	Data

图1：cache block

3.2 CMU

CMU实质上是把cache中的状态机部分和与CPU, Memory的交互部分独立出来，作为一个控制单元，控制数据的处理。

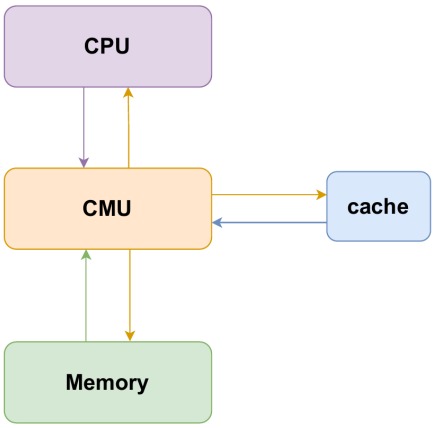


图2：架构

CMU的状态机：

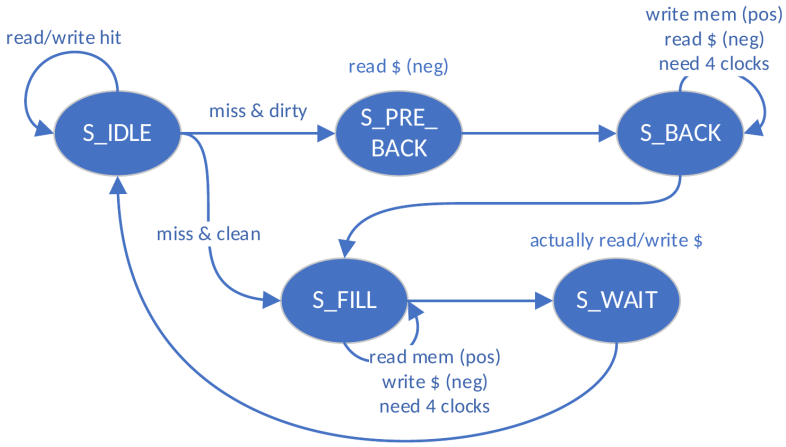


图3：CMU状态机

CMU的状态机为5个状态，各个状态的功能为描述如下：

1. **S_IDLE:** cache正常读写，即load/store命中或者未使用cache。
2. **S_PRE_BACK:** 为了写回，先进行一次读cache
3. **S_BACK:**上升沿将上个状态的数据写回到memory，下降沿从cache读下次需要写回的数据（因此最后一次读无意义），由计数器控制直到整个cache block全部写回。由于memory设置为4个周期完成读写操作，因此需要等待memory给出ack信号，才能进行状态的改变。
4. **S_FILL:**上升沿从memory读取数据，下降沿向cache写入数据，由计数器控制直到整个cache block全部写入。与S_BACK类似，需要等待ack信号。
5. **S_WAIT:** 执行之前由于miss而不能进行的cache操作。

4 实验要求

1. cache和CMU要求采取write-back, write-allocate,LRU的策略
2. 通过仿真测试和上板验证

5 实验步骤

5.1 补全cache和CMU模块的代码

（1） `cmu.v`

```
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

// next state logic
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

            // prepare to write back
            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; //done
                else
                    next_state = S_BACK;
```

```
                    if (mem_ack_i)
                        next_word_count = word_count + 1;
                    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 + 1;
                    else
                        next_word_count = word_count;
                end

                // respond eventually
                // seems that it can be optimized
                S_WAIT: begin
                    next_state = S_IDLE;
                    next_word_count = 2'b00;
                end
            endcase
        end
    end

    // cache ctrl ...
    // mem ctrl ...
    assign mem_data_o = cache_dout;

    assign stall = (next_state != S_IDLE); //state ?? next state ??
```

(2) `cache.v`

```
    assign addr_tag = addr[31:(32-TAG_BITS)];
    assign addr_index = addr[(31-TAG_BITS):(WORD_BYTES_WIDTH+ELEMENT_WORDS_WIDTH)];
    assign addr_element1 = {addr_index, 1'b0};
    assign addr_element2 = {addr_index, 1'b1};
    assign addr_word1 = {addr_element1,
addr[ELEMENT_WORDS_WIDTH+WORD_BYTES_WIDTH-1:WORD_BYTES_WIDTH]};
    assign addr_word2 = {addr_element2,
addr[ELEMENT_WORDS_WIDTH+WORD_BYTES_WIDTH-1:WORD_BYTES_WIDTH]};           //need to
fill in

    assign word1 = inner_data[addr_word1];
    assign word2 = inner_data[addr_word2];
    assign half_word1 = addr[1] ? word1[31:16] : word1[15:0];
    assign half_word2 = addr[1] ? word2[31:16] : word2[15:0];
    assign byte1 = addr[1] ?
                   addr[0] ? word1[31:24] : word1[23:16] :
                   addr[0] ? word1[15:8] :  word1[7:0]   ;
    assign byte2 = addr[1] ?
                   addr[0] ? word2[31:24] : word2[23:16] :
                   addr[0] ? word2[15:8] :  word2[7:0]   ;
```

```
    assign recent1 = inner_recent[addr_element1];
    assign recent2 = inner_recent[addr_element2];              //need to fill in
    assign valid1 = inner_valid[addr_element1];
    assign valid2 = inner_valid[addr_element2];               //need to fill in
    assign dirty1 = inner_dirty[addr_element1];
    assign dirty2 = inner_dirty[addr_element2];               //need to fill in
    assign tag1 = inner_tag[addr_element1];
    assign tag2 = inner_tag[addr_element2];                  //need to fill in

    assign hit1 = valid1 & (tag1 == addr_tag);
    assign hit2 = valid2 & (tag2 == addr_tag);                //need to fill in

    always @ (posedge clk) begin

        // hit?
        hit <= hit1 || hit2;

        // info about to-be-replaced cache line
        if (recent1) begin
            tag <= tag2;
            dirty <= dirty2;
            valid <= valid2;
        end
        else begin
            tag <= tag1;
            dirty <= dirty1;
            valid <= valid1;
        end

        // dout
        if (load) begin
            if (hit1) begin
                dout <=
                    u_b_h_w[1] ? word1 :
                    u_b_h_w[0] ? {u_b_h_w[2] ? 16'b0 : {16{half_word1[15]}},
half_word1} :
                    {u_b_h_w[2] ? 24'b0 : {24{byte1[7]}}, byte1};
            end
            else if (hit2) begin
                dout <=
                    u_b_h_w[1] ? word2 :
                    u_b_h_w[0] ? {u_b_h_w[2] ? 16'b0 : {16{half_word2[15]}},
half_word2} :
                    {u_b_h_w[2] ? 24'b0 : {24{byte2[7]}}, byte2};
            end
        end
        else dout <= inner_data[ recent1 ? addr_word2 : addr_word1 ];

        // read $ with load==0 means moving data from $ to mem
        // no need to update recent bit
        // otherwise the refresh process will be affected
        // recent
        case ({load, edit, store, invalid})
            //load & edit
            4'b1000, 4'b0100: begin
```

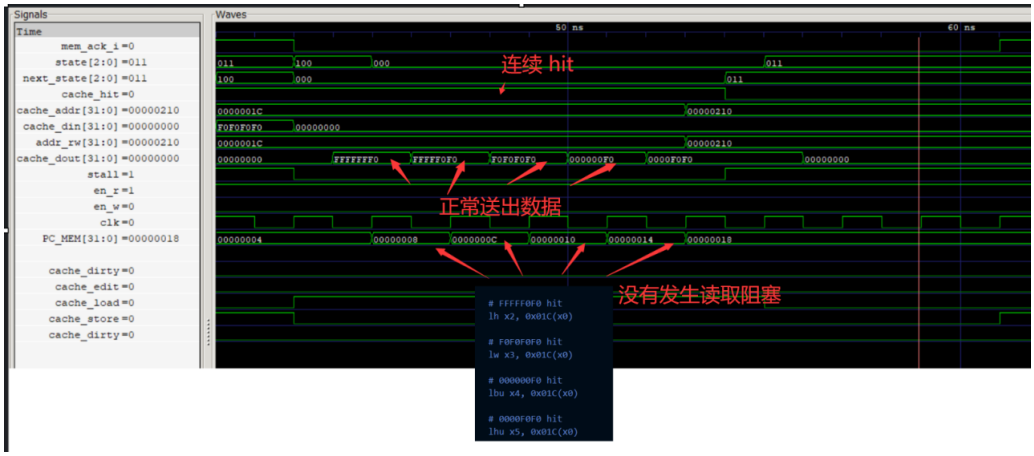
```
                if (hit1) begin
                    inner_recent[addr_element1] <= 1'b1;
                    inner_recent[addr_element2] <= 1'b0;
                end
                else if (hit2) begin
                    inner_recent[addr_element2] <= 1'b1;
                    inner_recent[addr_element1] <= 1'b0;
                end
            end
            4'b0001: begin
                inner_recent[addr_element1] <= 1'b0;
                inner_recent[addr_element2] <= 1'b0;
            end
            default: begin end
        endcase

        // dirty
        case ({load, edit, store, invalid})
            4'b0100: begin
                if (hit1) begin
                    inner_dirty[addr_element1] <= 1'b1;
                end
                else if (hit2) begin
                    inner_dirty[addr_element2] <= 1'b1;
                end
            end
            4'b0010: begin
                if (recent1) // replace 2
                    inner_valid[addr_element2] <= 1'b1;
                else inner_valid[addr_element1] <= 1'b1;
            end
            4'b0001: begin
                inner_dirty[addr_element1] <= 1'b0;
                inner_dirty[addr_element2] <= 1'b0;
            end
            default: begin end
        endcase

        // valid
        case ({load, edit, store, invalid})
            4'b0010: begin
                if (recent1) // replace 2
                    inner_dirty[addr_element2] <= 1'b0;
                else inner_dirty[addr_element1] <= 1'b0;
            end
            4'b0001: begin
                inner_valid[addr_element1] <= 1'b0;
                inner_valid[addr_element2] <= 1'b0;
            end
            default: begin end
        endcase

        // tag
        case ({load, edit, store, invalid})
            4'b0010: begin
                if (recent1) // replace 2
```

```
                inner_tag[addr_element2] <= addr_tag;
            else inner_tag[addr_element1] <= addr_tag;
        end
        default: begin end
    endcase

    // data
    case ({load, edit, store, invalid})
        4'b0100: begin
            if (hit1) begin
                inner_data[addr_word1] <=
                    u_b_h_w[1] ?        // word?
                        din
                    :
                        u_b_h_w[0] ?    // half word?
                            addr[1] ?       // upper / lower?
                                {din[15:0], word1[15:0]}
                            :
                                {word1[31:16], din[15:0]}
                        :   // byte
                            addr[1] ?
                                addr[0] ?
                                    {din[7:0], word1[23:0]}   // 11
                                :
                                    {word1[31:24], din[7:0], word1[15:0]} // 10
                            :
                                addr[0] ?
                                    {word1[31:16], din[7:0], word1[7:0]}   // 01
                                :
                                    {word1[31:8], din[7:0]} // 00
                ;
            end
            else if (hit2) begin
                inner_data[addr_word2] <=
                    u_b_h_w[1] ?        // word?
                        din
                    :
                        u_b_h_w[0] ?    // half word?
                            addr[1] ?       // upper / lower?
                                {din[15:0], word2[15:0]}
                            :
                                {word2[31:16], din[15:0]}
                        :   // byte
                            addr[1] ?
                                addr[0] ?
                                    {din[7:0], word2[23:0]}   // 11
                                :
                                    {word2[31:24], din[7:0], word2[15:0]} // 10
                            :
                                addr[0] ?
                                    {word2[31:16], din[7:0], word2[7:0]}   // 01
                                :
                                    {word2[31:8], din[7:0]} // 00
                ;
            end
        end
```

```
            4'b0010: begin
                if (recent1) // replace 2
                    inner_data[addr_word2] <= din;
                else inner_data[addr_word1] <= din;
            end
            default: begin end
        endcase
    end

endmodule
```

6 思考题

1. 在实验报告分别展示缓存命中、不命中的波形，分析时延差异。



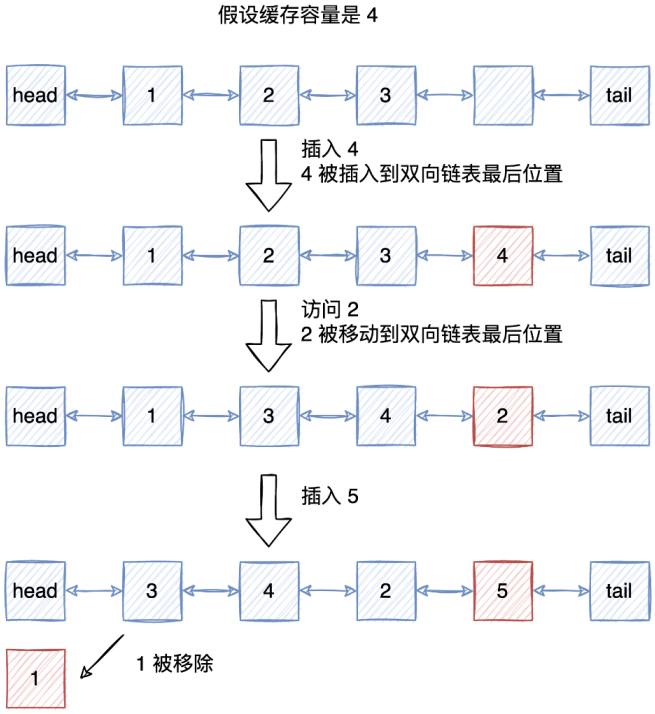
2. 在本次实验中，cache采取的是2路组相联，在实现LRU替换的时候，每一个set需要用多少bit来用于真正的LRU替换实现？

每一个缓存行需要多一个**LRU**位，而每一个组有两行，所以共需要多两位用于**LRU**实现。

3. 如果要实现cache的4路组相联，请描述一种真LRU和一种Pseudo-LRU的实现方式，并给出实现过程中每一个set需要用到多少bit来实现LRU。关于Pseudo-LRU，实现方式可以在网上查阅。

   - **LRU**

     需要维护一个 **set** 里每一行的排名，一共需要 $4 \times \log_2(4) = 8\text{bit}$ 来实现 **LRU**。维护排名的具体实现类似一个双向链表的插入和移除操作：

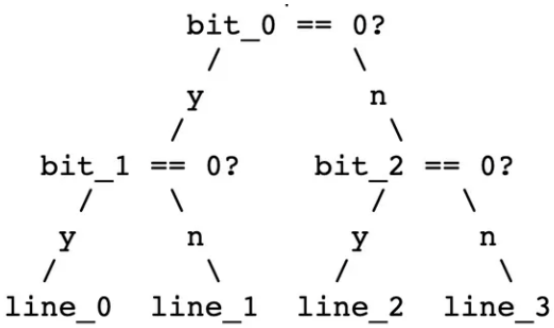


     只需要在每次访存操作之后，更新每一行被访问的时间排名即可。

   - **PLRU**

     PLRU 并不直接维护真实的 **LRU** 行，而是使用类似线段树的方式（**tree-PLRU**）来找到最有可能被替换的那一行。一个 **set** 只需要 **3 bit** 来维护。

     3 bit 中，第一个 **bit** 表示 **{line1, line2}, {line3, line4}** 最近被访问过的属于哪一组，第二个 **bit** 表示 **line1** 和 **line2** 哪个最迟被访问，第三个 **bit** 表示 **line3** 和 **line4** 哪个最迟被访问。如果第一个 **bit** 为**1**，则从 **{line3, line4}** 寻找替换行，否则，从 **{line1, line2}**，在分别根据第**2**，**3**个**bit**来确定替换行的行号。


bit_0 == 0?
y
n
bit_1 == 0?
bit_2 == 0?
y
n
y
n
line_0
line_1
line_2
line_3