实验十五报告

学号 2017K8009929044
2017K8009929025
姓名 李昊宸 李颖彦
箱子号 33

一、实验任务（10%）

1. 掌握 Cache 的知识
2. 学会设计 Cahe 模块

二、实验设计（80%）

（一）总体设计思路

Cache 的功能操作分为读操作和写操作。

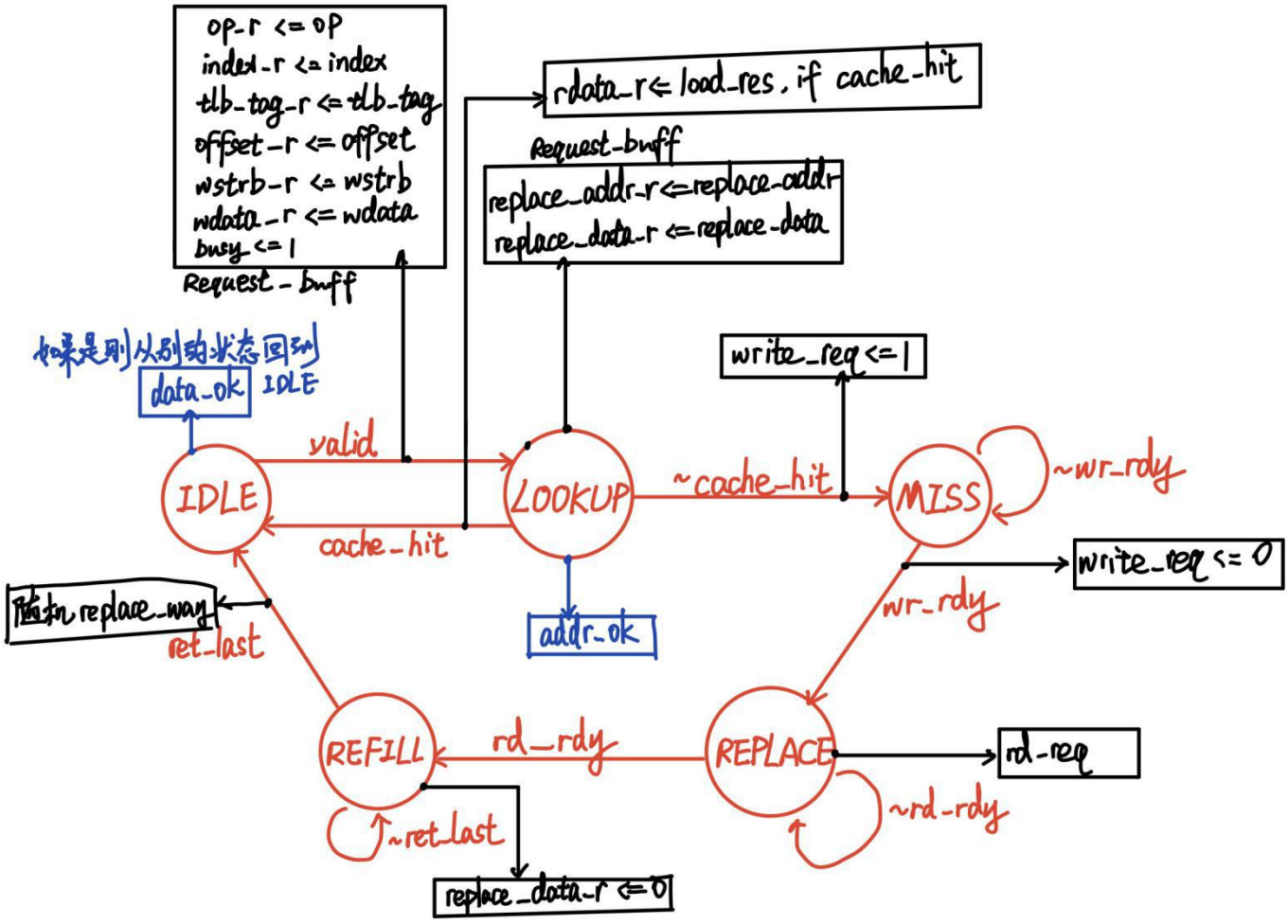
先来看读操作。将请求中虚地址的[11:4]位作为索引值（index）送往 Cache，将两路 Cache 中对应同一 index 的两个 Cache 行都读出。与此同时，将读操作的虚地址的[31:12]送往 TLB 模块查找得到物理地址。将 Cache 中读出的两个 Cache 行中的 Tag 信息与 TLB 查找转换出来的物理地址的[31:12]进行相等比较。如果某个 Cache 行的 Tag 比较相等，且该 Cache 行的有效位 V 等于 1，则表示访问命中在这个 Cache 行上。在进行 Tag 比较的同时，可以根据虚地址的[3:2]位对两个 Cache 行的 Data 信息进行选择，得到访问所在的 32 位数据 21。最后，根据 Tag 比较结果将命中那一路的 32 位数据返回。如果没有命中的 Cache 行，则需要通过总线接口向外发起访存请求，等到访存结果返回到 Cache 模块后，从返回结果中取出访问所在的 32 位数据，将其返回。

再来看写操作。写操作前面的操作步骤与读操作基本一致，区别仅在于写操作一上来可以不读取 Cache 的 Data 信息。它只需要读取两个 Cache 行中的 Tag、V 信息来判断 Cache 是否命中。如果 Cache 命中，则将待写的数据写入到命中的那个 Cache 行的对应位置上，同时将这一 Cache 行的脏位 D 置为 1。由于写操作有字节、半字、三字节、四字节不同的位宽，所以仅有地址信息还不够，还需要有写操作类型的信息予以配合。如果 Cache 缺失，由于我们是写回写分配的 Cache，所以我们要像读操作发生 Cache 缺失那样，先通过总线向外发起访存请求，然后等到访存结果返回 Cache 模块，最后将 store 要写的数据和内存重填的数据拼合在一

起，一并写入 Cache 中。

（二）重要模块 1 设计：cache 状态机

1、 工作原理



图一 cache 模块的状态机

2、 功能描述

初始状态为 IDLE，addr_ok 与 data_ok 均为 0。当接收到 valid 信号后，状态将转移到 LOOKUP，与此同时将端口处传入的数据存入 Request_buff。在 LOOKUP 发出 addr_ok，向 Request_buff 内存入替换行信息，通过 Request_buff 内的数据判断是否命中，如果命中就转移回 IDLE，并在回到 IDLE 的第一拍发出 data_ok。如果未命中，状态将转移到 MISS，同时 wr_req 信号拉高。在 MISS 阶段自旋等待 wr_rdy 信号。wr_rdy 信号是写地址和写数据 ok 信号（转接桥类 SRAM 的 addr_ok），也可以是写操作完成的 ok 信号（转接桥类 SRAM 的 data_ok），当该信号拉高时，状态将转移到 REPLACE，同时拉低 wr_req。当状态在 REPLACE 时拉高 rd_req，自旋等待 rd_rdy 信号。rd_rdy 信号是读地址 ok 信号（转接桥类 SRAM 的 addr_ok），当它拉高时状态将转移到 REFILL。REFILL 阶段自旋等待 ret_last 信号，期间不断接收从转接桥返回的读数据。ret_last 信号拉高表示最后一个读数据到达，状

态将转移到 IDLE，同时随机生成一个新的替换路 replace_way。

总体来说，读操作和写操作的状态转移图完全一致。读和写的唯一区别就是当命中时，写操作会修改 cache 内部对应行对应字节位置的数据；当未命中时，读会替换整个 cache 行，而写会将需要写的内容与从总线读回来的数据覆写后替换整个 cache 行。

3、 具体代码实现

三段式状态机：

```
//state machine
    reg [2:0] curstate;
    reg [2:0] nxtstate;
    parameter IDLE    = 3'd0;
    parameter LOOKUP  = 3'd1;
    parameter MISS  = 3'd2;
    parameter REPLACE = 3'd3;
    parameter REFILL   = 3'd4;

    always@(posedge clk) begin
        if(~resetn) begin
            curstate <= IDLE;
        end
        else begin
            curstate <= nxtstate;
        end
    end
//STATE TRANSFORMATION
    always@(*)
begin
    case(curstate)
        IDLE:
        begin
            if(valid)
                nxtstate = LOOKUP;
            else
                nxtstate = curstate;
        end
        LOOKUP:
        begin
            if(cache_hit)
                nxtstate = IDLE;
            else
                nxtstate = MISS;
        end
```

```
            MISS:
            begin
                if(wr_rdy)
                    nxtstate = REPLACE;
                else
                    nxtstate = curstate;
            end
            REPLACE:
            begin
                if(rd_rdy)
                    nxtstate = REFILL;
                else
                    nxtstate = curstate;
            end
            REFILL:
            begin
               if(ret_last)
                    nxtstate = IDLE;
                else
                    nxtstate = REFILL;
            end
            default:
                nxtstate = IDLE;
        endcase
    end
```

（三）重要模块 2 设计：cache 内部寄存器 Requset_buff

1、 功能描述

Request buff 用于记录访问 Cache 的操作类型、地址、字节写使能和写数据。这些信息均来自于 Cache 模块的接口。此外，Request buff 还要记录缺失 Cache 行准备要替换的路信息，以及已经从 AXI 总线返回了几个 32 位数据。

2、 具体代码实现

```
    //Request buffer
        reg           op_r;//1 write 0 read
        reg   [ 7:0]  index_r; //addr[11:4]
        reg   [19:0]  tlb_tag_r; //pfn + addr[12]
```

```
reg   [ 3:0]  offset_r; //addr[3:0]
reg   [ 3:0]  wstrb_r;
reg   [31:0]  wdata_r;
reg           busy;
reg   [127:0] replace_data_r;
always@(posedge clk)begin
    if(!resetn) begin
        op_r <= 0;
        index_r <= 0;
        tlb_tag_r <= 0;
        offset_r <= 0;
        wstrb_r <= 0;
        wdata_r <= 0;
        busy <= 0;
    end
    if(busy & data_ok) begin
        op_r <= 0;
        index_r <= 0;
        tlb_tag_r <= 0;
        offset_r <= 0;
        wstrb_r <= 0;
        wdata_r <= 0;
        busy <= 0;
    end
    if(curstate == IDLE & valid) begin
        op_r <= op;
        index_r <= index;
        tlb_tag_r <= tlb_tag;
        offset_r <= offset;
        wstrb_r <= wstrb;
        wdata_r <= wdata;
        busy <= 1;
    end
end

reg          replace_way;
//assign       replace_way = 0;

wire [127:0]  replace_data;
reg  [ 22:0]  pseudo_random_23;
//LSFR
always @ (posedge clk)
begin
       if (!resetn) begin
           pseudo_random_23 <= {7'b1010101,16'h00FF};
```

```
                replace_way <= 0;
            end
            else
                pseudo_random_23 <= {pseudo_random_23[21:0],pseudo_random_23[22] ^
pseudo_random_23[17]};

            if(curstate == REFILL & ret_last)
                replace_way <= pseudo_random_23[0];
    end
    //assign       replace_way = pseudo_random_23[0];

    always@(posedge clk)begin
        if(!resetn) begin
            replace_data_r <= 0;
        end
        if(curstate == LOOKUP) begin
            replace_data_r <= replace_data;
        end
        else if(curstate == REFILL) begin
            replace_data_r <= 0;
        end
    end
    wire [19:0]    replace_addr;
    reg  [19:0]    replace_addr_r;
    always@(posedge clk)begin
        if(!resetn) begin
            replace_addr_r <= 0;
        end
        if(curstate == LOOKUP) begin
            replace_addr_r <= replace_addr;
        end
        else if(curstate == REFILL) begin
            replace_addr_r <= 0;
        end
    end
 reg [1:0] rd_cnt;
    always @(posedge clk) begin
        if(!resetn) begin
            rd_cnt <= 2'b00;
        end
        else if(ret_valid) begin
            rd_cnt <= rd_cnt + 2'b01;
        end
    end
```

（四）重要模块 3 设计：cache 内部寄存器 二维 cache 表

1、 功能描述

（1） {Tag,Valid} RAM：调用 256 项*21 比特的 RAM，共需要 2 块 RAM。

（2）{Dirty} RAM：使用 regfiles 写法实现，实现为 2 个 256 项*1 比特的结构，或 1 个 256 项*2 比特的结构。

（3）{Data} RAM：调用 256 项*32 比特的 RAM，共需要 8 块 RAM。

2、 具体代码实现

每种类型仅拿一个作为举例

{Tag,Valid}RAM：

```
//tag_v_ram_0
    wire [2:0] tag_v_ram_0_we;
    wire [7:0] tag_v_ram_0_addr;
    wire [23:0]tag_v_ram_0_wdata;
    wire [23:0]tag_v_ram_0_rdata;

    tag_v_ram_0 my_tag_v_ram_0(
  .clka(clk),    // input wire clka
  .wea(tag_v_ram_0_we),      // input wire [2 : 0] wea
  .addra(tag_v_ram_0_addr),  // input wire [7 : 0] addra
  .dina(tag_v_ram_0_wdata),    // input wire [23 : 0] dina
  .douta(tag_v_ram_0_rdata)  // output wire [23 : 0] douta
);
    assign tag_v_ram_0_addr = busy? index_r : valid? index :0;
    assign tag_v_ram_0_wdata = {tlb_tag_r,4'b0001};
    assign tag_v_ram_0_we = (curstate == REFILL & replace_way == 0)? 3'b111:0;
```

{Dirty} RAM：

```
    //dirty_ram_0
    wire [7:0]   dirty_ram_0_raddr;
    wire         dirty_ram_0_rd;
    wire         dirty_ram_0_we;
    wire [7:0]   dirty_ram_0_waddr;
    wire         dirty_ram_0_wd;

    dirty_ram dirty_ram_0(
    .clk(clk),
    .resetn(resetn),
```

```
    // READ PORT
    .raddr(dirty_ram_0_raddr),
    .rdata(dirty_ram_0_rd),
    // WRITE PORT
    .we(dirty_ram_0_we),       //write enable, HIGH valid
    .waddr(dirty_ram_0_waddr),
    .wdata(dirty_ram_0_wd)
);
    assign dirty_ram_0_raddr = busy?index_r: valid? index: 0;
    assign dirty_ram_0_waddr = busy?index_r: valid? index: 0;
    assign dirty_ram_0_wd = (op_r == 1);
    assign dirty_ram_0_we = (curstate == LOOKUP & way0_hit & op_r == 1) | (cursta
te == REFILL & replace_way == 0 & op_r == 1);

module dirty_ram(
    input         clk,
    input         resetn,
    // READ PORT
    input  [ 7:0] raddr,
    output        rdata,
    // WRITE PORT
    input         we,       //write enable, HIGH valid
    input  [ 7:0] waddr,
    input         wdata
);
reg [255:0] rf;
//WRITE
always @(posedge clk) begin
    if(!resetn)
        rf <= 0;
    else if (we) rf[waddr]<= wdata;
end
//READ OUT 1
assign rdata = rf[raddr];
endmodule
```

{Data} RAM：

```
//data_ram_bank_0/////////////////////////////////////////////////
    wire [3:0] data_ram_bank0_0_we;
    wire [7:0] data_ram_bank0_0_addr;
    wire [31:0]data_ram_bank0_0_wdata;
    wire [31:0]data_ram_bank0_0_rdata;
    data_ram_bank0_0 my_data_ram_bank0_0(
  .clka(clk),    // input wire clka
```

```
  .wea(data_ram_bank0_0_we),      // input wire [3 : 0] wea
  .addra(data_ram_bank0_0_addr),  // input wire [7 : 0] addra
  .dina(data_ram_bank0_0_wdata),    // input wire [31 : 0] dina
  .douta(data_ram_bank0_0_rdata)  // output wire [31 : 0] douta
);
    assign data_ram_bank0_0_we =  (curstate == LOOKUP & cache_hit & way0_hit &off
set_r[3:2] == 2'b00 & op_r == 1)?wstrb_r://hit store
                                   (curstate == REFILL & rd_cnt == 2'b00 & ret_va
lid & replace_way == 0)?4'b1111://refill
                                   0;
    assign data_ram_bank0_0_addr = (curstate == IDLE)?index:index_r;
    assign data_ram_bank0_0_wdata = (curstate == LOOKUP & cache_hit & offset_r[3:
2] == 2'b00)?wdata_r://hit store
                                     (curstate == REFILL)? (offset_r[3:2] == 2'b0
0)? (wstrb_r == 4'b1111)?wdata_r:

    (wstrb_r == 4'b1110)?{wdata_r[31:8],ret_data[7:0]}:

    (wstrb_r == 4'b1100)?{wdata_r[31:16],ret_data[15:0]}:

    (wstrb_r == 4'b1000)?{wdata_r[31:24],ret_data[23:0]}:

    (wstrb_r == 4'b0000)?ret_data:

    (wstrb_r == 4'b0001)?{ret_data[31:24],wdata_r[7:0]}:

    (wstrb_r == 4'b0011)?{ret_data[31:16],wdata_r[15:0]}:

    (wstrb_r == 4'b0111)?{ret_data[31:24],wdata_r[23:0]}:

    wdata_r:
                                                          ret_data:
                                     0;
```

（五）重要模块 4 设计：cache 内部其他控制逻辑

1、 功能描述

返回 CPU 端的输出信号：

addr_ok ： 该次请求的地址传输 OK，读：地址被接收；写：地址和数据被接收

data_ok ： 该次请求的数据传输 OK，读：数据返回；写：数据写入完成

rdata ：读 Cache 的结果

返回转接桥端的输出信号：

rd_req ：读请求有效信号。高电平有效

rd_type ：读请求类型。3’b000——字节，3’b001——半字，3’b010——字，3’b100—— Cache 行。

rd_addr ：读请求起始地址

wr_req ：写请求有效信号。高电平有效。

wr_type ：写请求类型。3’b000——字节，3’b001——半字，3’b010——字，3’b100—— Cache 行。

wr_addr ：写请求起始地址

wr_wstrb：写操作的字节掩码。仅在写请求类型为 3’b000、3’b001、3’b010 情况下才有意义。

wr_data ：写数据

2、 具体代码实现

3、

```
assign addr_ok = curstate == LOOKUP;
assign data_ok = curstate == IDLE & start;

assign rd_req = curstate == REPLACE;
assign rd_type = 3'b100;  //REPLACE CACHE-ROW ONLY
assign rd_addr = {tlb_tag_r,index_r,4'b00};

reg write_req;
always@(posedge clk) begin
    if(!resetn)
        write_req <= 0;
    else if(curstate == LOOKUP & nxtstate == MISS & ((dirty_ram_1_rd == 1)&re
place_way | (dirty_ram_0_rd == 1)&~replace_way))
        write_req <= 1;
    else if(wr_rdy)
        write_req <= 0;
End

assign wr_req = write_req;
assign wr_type = 3'b100;//REPLACE CACHE-ROW ONLY
assign wr_addr = {replace_addr_r,index_r,4'b00};
assign wr_wstrb = 4'b1111;//nonsense
assign wr_data = replace_data_r;

reg [31:0] rdata_r;

always@(posedge clk) begin
    if(!resetn) begin
```

```
            rdata_r <= 0;
        end
        else if(curstate == LOOKUP & cache_hit)
            rdata_r <= load_res;

        else if(offset_r[3:2] == 2'b00 & rd_cnt == 2'b00 & ret_valid)
            rdata_r <= ret_data;
        else if(offset_r[3:2] == 2'b01 & rd_cnt == 2'b01 & ret_valid)
            rdata_r <= ret_data;
        else if(offset_r[3:2] == 2'b10 & rd_cnt == 2'b10 & ret_valid)
            rdata_r <= ret_data;
        else if(offset_r[3:2] == 2'b11 & rd_cnt == 2'b11 & ret_valid)
            rdata_r <= ret_data;
    end
    assign rdata = rdata_r;

    assign way0_hit = way0_v && (way0_tag == tlb_tag_r & curstate != IDLE);
    assign way1_hit = way1_v && (way1_tag == tlb_tag_r & curstate != IDLE);

    assign cache_hit = way0_hit || way1_hit;

    assign replace_addr = replace_way? way1_tag : way0_tag;

    assign way0_load_word = (offset_r[3:2] == 2'd0)?data_ram_bank0_0_rdata:
                            (offset_r[3:2] == 2'd1)?data_ram_bank1_0_rdata:
                            (offset_r[3:2] == 2'd2)?data_ram_bank2_0_rdata:
                            data_ram_bank3_0_rdata;

    assign way1_load_word = (offset_r[3:2] == 2'd0)?data_ram_bank0_1_rdata:
                            (offset_r[3:2] == 2'd1)?data_ram_bank1_1_rdata:
                            (offset_r[3:2] == 2'd2)?data_ram_bank2_1_rdata:
                            data_ram_bank3_1_rdata;

    assign load_res = {32{way0_hit}} & way0_load_word
                    | {32{way1_hit}} & way1_load_word;

    assign replace_data = replace_way? {data_ram_bank3_1_rdata,data_ram_bank2_1_r
    data,data_ram_bank1_1_rdata,data_ram_bank0_1_rdata}:

                                        {data_ram_bank3_0_rdata,data_ram_bank2_0_
    rdata,data_ram_bank1_0_rdata,data_ram_bank0_0_rdata};
```

三、实验过程（10%）

（一）实验流水账

2019/12/18　22:00 – 24:00 代码书写

2019/12/19　00:00 – 16:00　代码书写+debug

2019/12/22　2:00　– 6:00 书写实验报告

（二）错误记录

1、错误 1：仿真逻辑与预期不符

（1）错误现象

根据测试文件的定义，应该是产生四个随机 tag 和 data 对，然后向 cache 内部写入数据，之后再进行读取比对。但是发生了在第四组的写 cache 过程中，只写了 cache 行中的前三个 32 位寄存器，第四个的写请求并没有发出就进入读取数据比对的阶段，从而触发错误。

（2）分析定位过程

跟踪仿真文件中的两个两位寄存器 count_i 和 count_j，发现在文件中记录的定义表示每次写一个寄存器，j 会加 1，当 j = 3 而且再加 1 时，i 会加 1。当 i 和 j 都重新变成 0 时，结束 write 阶段。但是，波形中显示 valid 再还没有拉高时，op 已经拉高，表示为写，并且 j 从初始值 0 变为 1。

道理上来说，j 只有在接受到 data_ok 信号时才会加 1。于是去检查 data_ok 的值，发现在 reset 之后，data_ok 就已经是 1 了。这与设计理念中 addr_ok 之后才会发 data_ok，并且 data_ok 只会维持一拍的方案不符。

（3）错误原因

起初的赋值语句为：*assign data_ok = curstate == IDLE;*

这显然是导致错误的原因，应该对该赋值做修改。

（4）修正效果

```
reg start;
always@(posedge clk) begin
    if(!resetn)
        start <= 0;
    else if(valid)
        start <= 1;
    else if(data_ok)
        start <= 0;
end
assign data_ok = curstate == IDLE & start;
```

修正后错误消失。

2、错误 2： cache 命中异常

（1）错误现象

当状态机重新回到 IDLE，并且立刻有新的 valid 信号输入时，way_0_hit 和 way_1_hit 信号会在状态转移到 LOOKUP 时发生改变。这是不被允许的错误。

（2）分析定位过程

问题应该是 way_0_hit 和 way_1_hit 的复制逻辑错误。原先的赋值语句（也就是根据讲义中的赋值语句进行寄存器化）是这样的：*assign way0_hit = way0_v && (way0_tag == tlb_tag_r);*

但是这样会导致一种情况发生：当 valid 持续有效时，在状态机回到 IDLE 时，tlb_tag_r 要等到下一拍到来才会更新，但是 way0_hit 此时已经做出判断，并修改了对应的 cache_hit 选项，很有可能导致未知错误。

（3）错误修改

将赋值语句进行如下修改：

```
assign way0_hit = way0_v && (way0_tag == tlb_tag_r & curstate != IDLE);
```

也就是始终让状态在 IDLE 时发出 cache 不命中的信号，这样可以让 cache 默认保存对应替换行的替换数据，避免了原先可能会发生的这种错误：IDLE 阶段 cache 命中，于是就没有保存替换行信息，但是到了 LOOKUP 阶段发现 cache 其实没有命中，此时错误的替换行信息会导致内存修改错误。