**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。

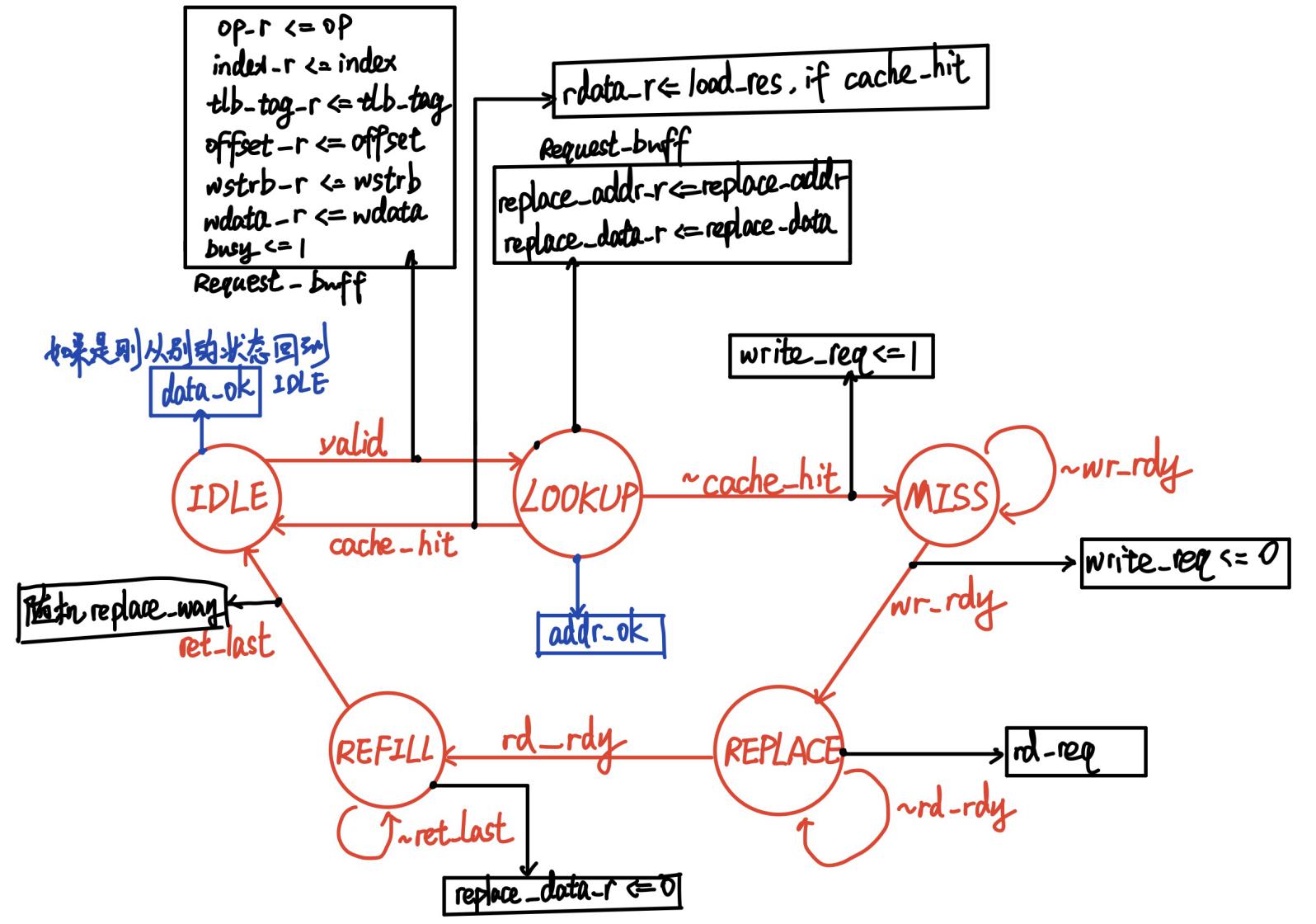
如果 Cache缺失，由于我们是写回写分配的 Cache，所以我们要像读操作发生 Cache 缺失那样，先通过总线向

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

1. 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。

1. 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。

1. 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
2. {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。

1. 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) | (curstate == 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 &offset\_r[3:2] == 2'b00 & op\_r == 1)?wstrb\_r://hit store

                                   (curstate == REFILL & rd\_cnt == 2'b00 & ret\_valid & 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'b00)? (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

1. 具体代码实现

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。