实验 16 报告

学号: 2018K8009929043 2018K8009929035
姓名: 曾冕 张翔雨
箱子号： 06

一、实验任务（10%）

本实验独立于之前的实验，设计了一个 2 路组相联，写回写分配的伪随机替换算法的非阻塞 cache。Lab15 完成了整体模块的设计，lab16 及之后的实验会将其综合进之前完成的 CPU 内。

二、实验设计（40%）

（一）总体设计思路

Cache 的内部逻辑用 1 个带有 5 个状态的状态机来控制，分别为 IDLE,LOOKUP , MISS , REPLACE , REFILL，基本对应了 cache 模块的四种访问模式（look up , hit write , replace, refill）。由于 cache 的基本原理就是一张二维查找表，本实验用了 3 种不同的 ram 来实现 cache 的查找功能，分别为 21*256 的{tag，v}表，1*256 的 dirty 表，和 32*256 的 data 表（需要 8 个，因为实现的是两路 4 行的 cache）。

Cache 内部的状态机转移逻辑基本同讲义手册种类似，和手册不一样的一点是在 lookup 阶段的自旋被统一归类到了返回到 idle 的状态转移，同时在 IDLE 阶段增加了 busy_tag 作为阻塞标记，具体实现方式见限免的详细设计部分。

一个简单的内部数据通路图如下图所示：

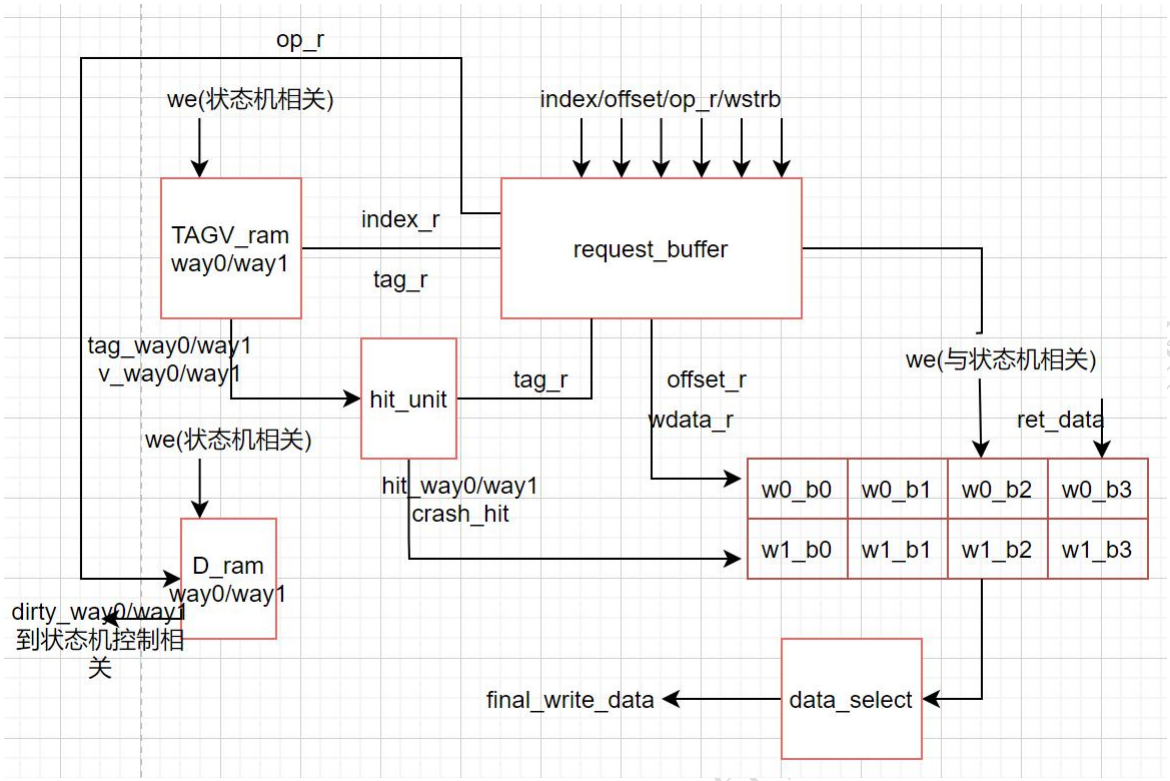

图 01: 整体 cache 内部数据通路设计

（二）重要模块 1 设计：cache 状态机

1、工作原理（内容描述）

根据外部的输入信号和内部的查找结果，实现 cache 状态的变更，阻塞可能的冲突。

2、接口定义（cache 的所有接口都在状态机中展示）

名称	方向	位宽	功能描述
clk	IN	1	时钟信号
Cpu 与 cache 的交互			
valid	IN	1	请求有效
op	IN	1	1：write 0:read
index	IN	8	地址 index 域
tag	IN	20	从 tlb 查到的 pfn 形成的 tag，由于来自组合逻辑运算，故与 index 是同拍信号。
offset	IN	4	地址的 offset
wstrb	IN	4	字节写使能
wdata	IN	32	写数据
Addr_ok	OUT	1	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	OUT	1	该次请求的数据传输 OK，读：数据返回；写：数据写入完成
Rdata	OUT	32	读 Cache 的结果
Cache 和转接桥的交互			
Rd_req	OUT	1	读请求有效信号。高电平有效。

名称	方向	位宽	功能描述
Rd_type	OUT	3	读请求类型,本次为 100
Rd_addr	OUT	32	读请求起始地址
Rd_rdy	IN	1	读请求能否被接收的握手信号。高电平有效。
Ret_valid	IN	1	返回数据有效
Ret_last	IN	2	物返回数据是最后一个数据的标志（burst 相关）
Ret_data	IN	32	读返回数据
Cache 写交互			
Wr_req	OUT	1	写请求有效
Wr_type	OUT	3	写请求类型 本次为 100
Wr_addr	OUT	32	写请求起始地址
Wr_wstrb	OUT	4	写操作的字节掩码
Wr_data	OUT	128	写数据
Wr_rdy	IN	3	写请求能否被接收的握手信号。高电平有效。此处要求 wr_rdy 要先于 wr_req 置起，wr_req 看到 wr_rdy 后才可能置上。

3、 功能描述

用图片的展示形式如下：

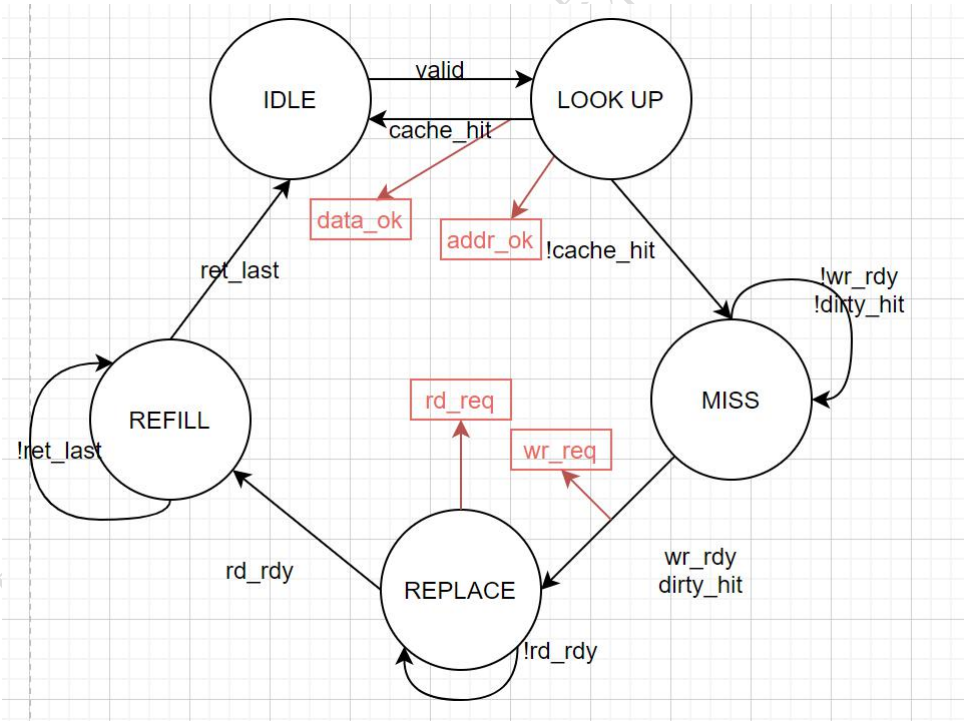

图 02:：cache 状态机的设计

基本设计逻辑同讲义，除了 look_up 状态机中使用了 cache_hit 返回到 idle 而不是自旋等待，以及 miss->replace 状态机转换时增加了是否为脏的判断。其中涉及到输出信号的部分已经在图中用带框的部分标出，具体的代码如下所示：

最为关键的 data_ok 和 addr_ok 信号如下：

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

图 03:：data_ok 的逻辑

Data_ok 的逻辑是先用一个 data_ok_valid 寄存器，存储一拍的有效信号，同时有效信号与状态机取与，当 valid 信号来的时候，下一拍就会进 LOOK UP，只有在命中返回的情况下才会再回到 IDLE 状态，如果此时状态处于 IDLE，说明命中了，这个时候自然 data_ok 拉高，同时再寄存器中拉低 data_ok_valid，保证 data_ok 只持续一拍，下一拍是新的 cache 有效信号。

Addr_ok 的逻辑逻辑由于不涉及自己的状态机改动，逻辑较为简单，如下代码所示：

```
assign addr_ok = (curstate == LOOKUP)|(curstate==IDLE & !valid);
```

wr_req 信号也与讲义处理方式相同，用一个寄存器暂存，在进行条件转换的时候自然拉高，当接收到总线返回的 ready 信号之后拉低。

```
always@(posedge clk) begin
    if(!resetn)
        wr_req_r <= 0;
    else if((curstate == MISS & nxtstate == REPLACE) & !wr_rdy)
    //else if((curstate == MISS & nxtstate == REPLACE) ) for bug commi
        wr_req_r <= 1;
    else if(wr_rdy)
        wr_req_r <= 0;
end
```

图 04:：wr_req_r 的逻辑代码

（三）重要模块 2 设计：cache 内次数据通路

1、工作原理

Cache 模块内除了表项以外的其他数据通路，主要包括 request_buffer，data slect， miss_buffer,LSFR 等主要功能部件

2、接口定义（见重要模块 1 设计这里不重复叙述）

3、功能描述

首先是 Request_buffer，作用是用于存储输入信息，在有效的情况下将内部信号存入，类似于之前的流水线处理器中的接受包过程。 代码如下：

```
always@(posedge clk)begin
    if(!resetn) begin
        Request_buffer <= 0;
        busy_tag <= 0;
    end
    if(busy_tag & data_ok) begin
        Request_buffer <= 0;
        busy_tag <= 0;
    end
    if(curstate == IDLE & valid) begin//来了一拍有效数据
        Request_buffer <= {op,index,tag,offset,wstrb,wdata};
        busy_tag <= 1;
    end
end
assign {
        op_r,
        index_r,
        tag_r,
        offset_r,
        wstrb_r,
        wdata_r
        }=Request_buffer;
```

图 05:requeset_buffer 的代码

另一个较为重要的部件 LSFR，利用线性位移方式生成位随机数，并将随机数最后一位作为 replace_way 信号，这里没有用与 cache_top 中的方式，因为根据实际波形图来看 cache_top 中的生成随机数方法并不是很随机，所以用了新的方法，会对仿真时长产生一定影响：

```
/************LSFR**************/
reg [15:0]  dout;
reg [15:0]  dout_next;
always @ (posedge clk )begin
    if(!resetn)       dout <=  INIT;
    else              dout <=  dout_next;
end
integer i;
always@(*)
begin
dout_next[0] <= dout[2];
for(i=1; i<16; i=i+1)
    if(COFF[16-i])        dout_next[i] <= dout[i-1]^dout[2];
    else                  dout_next[i] <= dout[i-1];
end
assign replace_way = dout[0];
/*************LSFR finish*************/
```

图 06:LFSR 的代码

MISS_buffer 没有严格拼成一个大数据，而是分别用了 data_buffer 和 rd_cnt 寄存器暂存发生 miss 后的信息，代码如下图所示：

```
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
reg [31:0] data_buffer;
always@(posedge clk) begin
    if(!resetn) begin
        data_buffer <= 0;
    end
    else if(curstate == LOOKUP & cache_hit)
        data_buffer <= load_data;
    else if(row_off == 2'b00 & rd_cnt == 2'b00 & ret_valid)
        data_buffer <= ret_data;
    else if(row_off == 2'b01 & rd_cnt == 2'b01 & ret_valid)
        data_buffer <= ret_data;
    else if(row_off == 2'b10 & rd_cnt == 2'b10 & ret_valid)
        data_buffer <= ret_data;
    else if(row_off == 2'b11 & rd_cnt == 2'b11 & ret_valid)
        data_buffer <= ret_data;
end
assign rdata = data_buffer;
```

图 07:类 miss_buffer 的结构代码

当返回数据有效时，cnt 寄存器会自加，记录返回的数据数目；在返回数据有效的情况下，当行偏移和返回数据个数相同时（即匹配上的时候），会将读返回数据赋值给 data_buffer

有关 cache 命中的逻辑处理如下：

```
assign way0_hit = way0_v && (way0_tag == tag_r );
assign way1_hit = way1_v && (way1_tag == tag_r );
assign cache_hit = way0_hit || way1_hit;
assign replace_addr = replace_way? way1_tag : way0_tag;
```

图 08:命中判断数据通路

（四）重要模块 3 设计：cache 表项管理

1、工作原理

Cache 模块内的主要数据通路，12 张 ram 表，具体分为哪些已经在总设计思路中提及，这里主要详细介绍逻辑。

2、接口定义（见重要模块 1 设计这里不重复叙述）

3、功能描述

首先是 tag_v 表，由于 tag 和 v 的高度重复性，这两个 cache 标志项被放在一张表内组织，使得查询和读写更高效，例化的 ip 核为 21*256，一个示例如下：

```
TAGV_ram my_Way0_TAGV(
    .clka(clk),    // input wire clka
    .wea(Way0_TAGV_we),      // 1
    .addra(Way0_TAGV_addr),  // [7:0]
    .dina(Way0_TAGV_wdata),  // [20:0]
    .douta(Way0_TAGV_rdata)  // [20:0]
);
```

图 09:调用 ip_catalog 的例化 tag_v_ram

其地址，写数据和使能信号的处理较为简单：

```
assign Way0_TAGV_addr   = busy_tag? index_r : valid? index :0;
assign Way1_TAGV_addr   = busy_tag? index_r : valid? index :0;
assign Way0_TAGV_wdata  = {tag_r,1'b1};
assign Way1_TAGV_wdata  = {tag_r,1'b1};
assign Way0_TAGV_we     = (curstate == REFILL & !replace_way)? 1'b1:0;
assign Way1_TAGV_we     = (curstate == REFILL & replace_way)? 1'b1:0;
```

图 10: 写数据和使能信号的处理

对于 dirty 项，考虑到之后可能会逻辑较为复杂，本实验中也例化了一个同步 ram，只是位宽设为 1，调用例化的例子同图 9，但 dirty 项的使能信号和写入数据较为巧妙，如下所示：

```
assign Way0_Dram_addr   = busy_tag?index_r: valid? index: 0;
assign Way1_Dram_addr   = busy_tag?index_r: valid? index: 0;
assign Way0_Dram_wdata  = (op_r == 1);
assign Way1_Dram_wdata  = (op_r == 1);
assign Way0_Dram_we     = (curstate == LOOKUP & way0_hit & op_r ) | (curstate == REFILL & !replace_way & op_r );
assign Way1_Dram_we     = (curstate == LOOKUP & way1_hit & op_r ) | (curstate == REFILL & replace_way & op_r );
```

图 11: dirty_ram 的控制相关信号

地址的选择相同，但写数据是仅在 cache 操作为写的时候才写入，同时在两个时间段都会对 dirty 位进行查询/写入，包括 REFILL(根据是否为脏来决定状态机跳转)和 LOOK UP（命中时要将对应的 d 位置 1）。

Bank_ram 的处理更加复杂，同时也是 cache 的主要实现方式。报告中将适度分块描述实现方式

对于写入 bank 的数据，可以进行预处理，代码如下所示：

```
wire [31:0] cache_write_data;
assign cache_write_data[31:24]  = wstrb_r[3] ? wdata_r[31:24]   : ret_data[31:24];
assign cache_write_data[23:16]  = wstrb_r[2] ? wdata_r[23:16]   : ret_data[23:16];
assign cache_write_data[15:8]   = wstrb_r[1] ? wdata_r[15:8]    : ret_data[15:8];
assign cache_write_data[7:0]    = wstrb_r[0] ? wdata_r[7:0]     : ret_data[7:0];
```

图 12:根据字节写使能生成预处理数据

使能，写入数据，地址等信息的控制如下所示（仅以第一路第一组为例）：

```
assign Way0_bank0_en    =   (curstate == LOOKUP & way0_hit &row_off == 2'b00 & op_r)?wstrb_r://hit store
                            (curstate == REFILL & rd_cnt == 2'b00 & ret_valid & !replace_way)?4'b1111:0;
assign Way0_bank0_addr  =   (curstate == IDLE)?index:index_r;
assign Way0_bank0_wdata =   (curstate == LOOKUP & cache_hit & row_off == 2'b00)?wdata_r://hit store
                            (curstate == REFILL)& (row_off == 2'b00)? cache_write_data:0;
```

图 13:bank_ram 控制信号

对于使能信号，当在 LOOK UP 阶段查找命中，且行偏移匹配后，就可以将对应的输入字节写使能信号直接拿来用，但如果是在替换的时候匹配上，由于替换的时候是整行的替换，所以需要将字节写使能强制变更为 1111。

对于地址，如果还在初始阶段就需要将原始输入赋值给地址，否则，就要选择 request_buffer 中在暂存的地址信息。

对于写数据，同样分为查找命中和不命中两种情况，当命中的时候就直接将写数据拿来用，否则，就需要用之前预处理的 cache_write_data 写入。

最后的输出的写数据的逻辑代码如下：

```
assign way0_load_word = ({32{offset_r == 4'b0000}} & Way0_bank0_rdata) |
                        ({32{offset_r == 4'b0100}} & Way0_bank1_rdata) |
                        ({32{offset_r == 4'b1000}} & Way0_bank2_rdata) |
                        ({32{offset_r == 4'b1100}} & Way0_bank3_rdata);
assign way1_load_word = ({32{offset_r == 4'b0000}} & Way1_bank0_rdata) |
                        ({32{offset_r == 4'b0100}} & Way1_bank1_rdata) |
                        ({32{offset_r == 4'b1000}} & Way1_bank2_rdata) |
                        ({32{offset_r == 4'b1100}} & Way1_bank3_rdata);
assign load_data      = {32{way0_hit}} & way0_load_word
                        |{32{way1_hit}} & way1_load_word;
assign replace_data   = replace_way? {Way1_bank3_rdata,Way1_bank2_rdata,Way1_bank1_rdata,Way1_bank0_rdata}:
                                     {Way0_bank3_rdata,Way0_bank2_rdata,Way0_bank1_rdata,Way0_bank0_rdata};
```

图 14:根据 bank_rdata 生成最后写数据输出的逻辑代码

这里讲义中都给出了实现方式，就不再多叙述

三、实验过程（50%）

（一）实验流水账

12.26 上午 8 点，开始干活，花费一上午写出了整个 cache 的框架，但仿真始终卡在第一个点，后来是发现例化的 bank_ram 没开启字节写使能。。

12.26 晚上 6 点，花费 4 小时，优化时序，让总仿真时间减少 30w。（发现伪随机算法还会影响到仿真时间）

12.27 开始写实验报告，零零碎碎大约花费 3 小时完成。 画图耗时约 1 小时，文字耗时大约 2 小时。

（二）错误记录

1、错误 1：data_ok 信号错误

（1） 错误现象

在第 00 个点的 cache 测试总是出错，错误代码为 cachers wrong，说明没有正确写入。

（2）分析定位过程

观察波形，发现是 data_ok 没有机会拉高，仿真无法进行，发现是由于状态机和讲义设计的不一样，所以导致照搬讲义的 data_ok 实现方式会出现由于 addr 没正常拉低而无法使得 data_ok 拉高的情况，

图 15:data_ok 一直为 0，addr_ok 一直为 1

（3）错误原因

状态机中没有实现 look_up 的自旋而是选择在 look_up 返回到 idle 态，所以对应的 data_ok 选项也要修改。考虑命中后返回的那一拍，由于进 look up 态会自然的输出一个 addr_ok，一个自然的想法就是在命中返回的那一拍输出一个 data_ok，这样就保证 data_ok 紧接这 addr_ok 下一拍输出。

（4）修正效果，

修改后代码见图 3 所示，修改后进入下一个错误记录点。

2、错误 2：wr_req 信号错误

（2） 错误现象

在第 00 个点的 replace 测试错误。

（2）分析定位过程

观察波形，发现是在 miss 向 replace 状态转换的时候异常拉高了，在 rdy 一直有效的情况下被置起（与讲义上的 icache 的设计中 req 一直为 0 矛盾，导致 replace 出错）

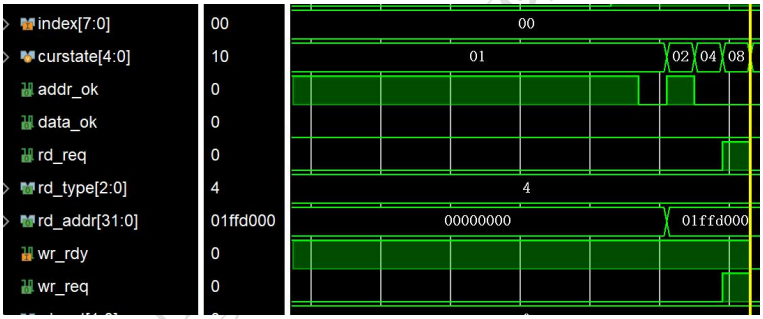

图 16:req 错误拉高

（3）错误原因

没有用 wr_rdy 有效来阻塞 wr_req 的转换，具体代码及修正方式见图 4。

（4）修正效果，

修正后正常运行到结束，上板测试通过

四、实验总结（可选）

本次实验本以为和之前的 TLB 实验一样的水，但是讲义的设计却很复杂，可能讲义试图让我们完成一个功能较为完善的 cache 块，所以很多在 lab16 中国并用不上的功能都要求我们添加，实际上 lab16 的仿真测试较为简单，不完全实现讲义手册的功能都可以通过。也可能是助教放我们一马，最后一个实验就要求不那么严了。