计算机体系结构实验

Lab 16 实验报告

小组成员：
王畅路 2019K8009907018
杜政坤 2019K8009909005

2021 年 12 月 20 日

1 实验任务

cache 实验只需要实现 lab16。lab16 是一个独立于其他环境的单独 cache。

本次实验要求设计一个 cache 模块，cache 容量为 8KB，为两路组相联，cache 行大小为 16 字节。要求采用 Tag 和 Data 同步访问的形式，“虚 Index 实 Tag” 的访问形式，伪随机替换算法，写回写分配策略，阻塞式设计，不采用 “关键字优先” 技术。在完成设计后，要求通过 cache 模块的针对测试并成功上板。由于 cache 还未集成进 CPU 中，需要对 cache 进行硬件初始化。

2 实验设计

2.1 总体设计思路

对于 cache 模块内部的数据通路，大致可以分为 cache 表和剩余模块两个部分。为了对这些模块的行为进行控制，引入了两个状态机，主状态机用于控制 cache 模块的查找及缺失处理操作，写状态机用于控制 cache 模块的写操作。

2.2 重要模块 1 设计：cache 表模块

1、工作原理

依据各种 Cache 访问执行的不同操作，将对 Cache 模块进行的访问归纳为四种：Look Up、Hit Write、Replace 和 Refill。四种访问定义如下：

Look Up：判断是否在 Cache 中，根据命中信息选取 Data 部分的内容。

Hit Write：命中的写操作会进入 Write Buffer，随后将数据写入到命中 Cache 行的对应位置。

Replace：为了给 Refill 的数据腾出地方而发起的读一整个 Cache 行的操作。

Refill：将内存返回的数据填入到 Replace 腾出来的位置上。

根据这些访问对 cache 的操作，可以将 cache 行信息分为 Tag 和 v、d、data 三组。TagV 实例化 2 块 256*21 的同步 RAM。为了尽可能让 data 能同时支持读写操作，每路 data 实例化 4 块 256*32 的同步 RAM，D 表实例化两块 256*1 的寄存器。

2、片选使能和写使能

		Look Up	Hit Write	Replace	Refill
{Tag,V}	片选	2 路	-	替换的那一路	替换的那一路
	写使能	-	-	-	替换的那一路
D	片选	-	写的那一路	替换的那一路	替换的那一路
	写使能	-	写的那一路	-	替换的那一路
Data	片选	2 路，1 个 bank	1 路，1 个 bank	替换的那一路的 4 个 bank	替换的那一路的 4 个 bank
	写使能	-	1 路，1 个 bank	-	替换的那一路的 4 个 bank

3、功能描述

以 cache 表的 data 表为例，实例化了 2 路 8 块 bank，每块 256*32。

```
    data_bank_ram Data_RAM_Way0_Bank0(
        .clka   (clk_g               ),
        .addra  (data_addr           ),
        .ena    (data_way0_bank0_en  ),
        .wea    (data_way0_bank0_we  ),
        .dina   (data_way0_bank0_din ),
        .douta  (data_way0_bank0_dout)
    );
```

tagv 表实例化了两块 ram，每块 256*21。

```
    tagv_ram TagV_RAM_Way0(
    .clka   (clk_g         ),
    .addra  (tagv_addr     ),
    .ena    (tagv_way0_en  ),
    .wea    (tagv_way0_we  ),
    .dina   (tagv_way0_din ),
    .douta  (tagv_way0_dout)
);
```

D 表用了两个 REGFILE

```
    //D reg way0/1
    reg D_Way0 [255:0];
    reg D_Way1 [255:0];
```

根据表 1，TagV 表的地址片选信号和写使能如下：

```
assign tagv_addr = (state == `IDLE && valid && addr_ok) ? index :
                   (state == `REFILL && recv_end) ? rb_index : 8'b0;
```

Data 表的地址片选信号和写使能如下：

```
assign data_addr = (wstate == `WRITE) ? wb_index :
                   (state == `IDLE && valid && addr_ok) ? index :
                   (state == `REFILL) ? rb_index : 8'b0;
```

2.3 重要模块 2 设计:cache 状态机模块

1. 工作原理

为了控制 cache 进行不同的操作，引入了主状态机和写状态机两个状态机。由于实现的是阻塞式 cache，Cache Miss 的时候不会接收新的请求，所以 Look Up 和 Replace 和 Refill 处理可以

共用一个状态机，称之为主状态机；Hit Write 是游离于 Look Up 和 Repalce 和 Refill 之外的单独访问，单独使用一个状态机维护，称之为写状态机。

2. 功能描述

主状态机共包括 5 个状态，其转换条件见图 6：IDLE：Cache 模块当前没有任何操作。

LOOKUP：Cache 模块当前正在执行一个操作且得到了它的查询结果。

MISS：Cache 模块当前处理的操作 Cache 缺失，且正在等待 AXI 总线的 wr_rdy 信号。

REPLACE：待替换的 Cache 行已经从 Cache 中读出，且正在等待 AXI 总线的 rd_rdy 信号。

REFILL：Cache 缺失的访存请求已发出，准备/正在将缺失的 Cache 行数据写入 Cache 中。

写状态机包含两个状态

IDLE：当前没有待写的数据。

WRITE：将待写数据正在写入到 Cache 中。

3. 状态转移图

主状态机和写状态机的状态转移图如下：

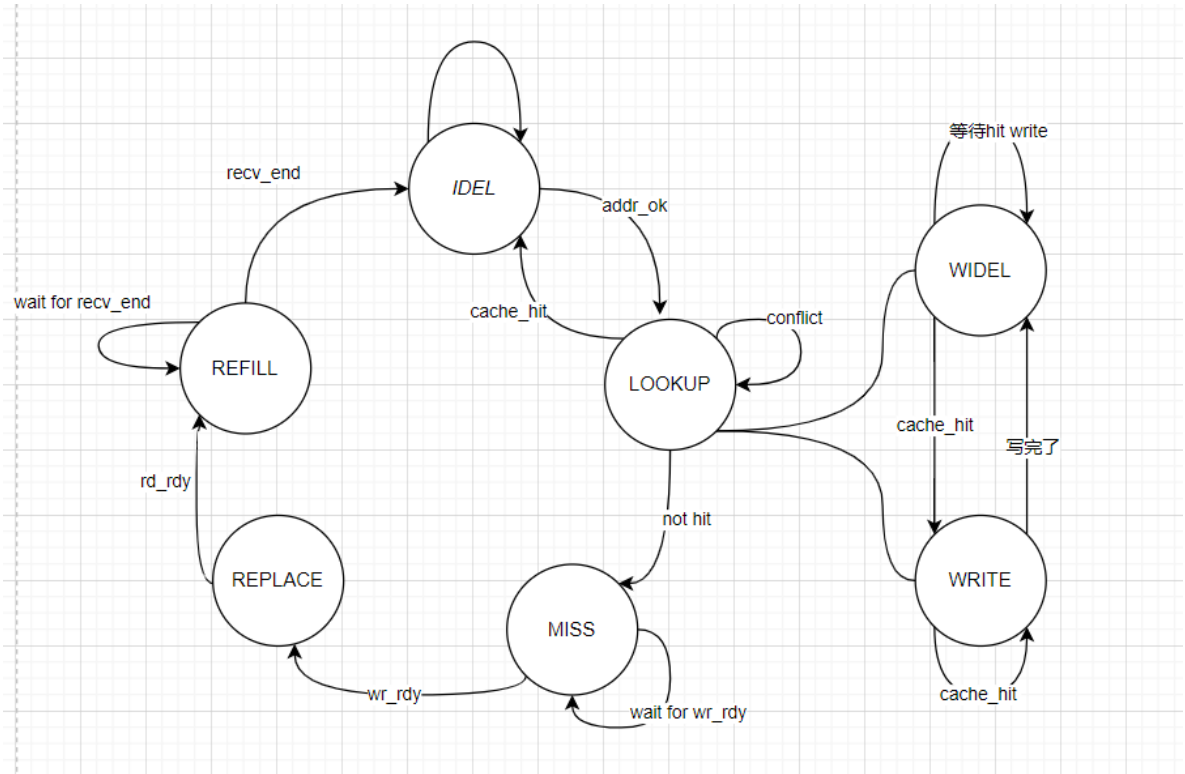

图 1: 总体逻辑框图

2.4 重要模块 3 设计：剩余模块

1. 工作原理

剩余的数据通路有 5 部分：Request Buffer、Tag Compare、Data Select、Miss Buffer、LFSR 和 Write Buffer。

Request Buffer 负责将 Cache 与 CPU 流水线的交互接口里的 op、index、tag、offset、wstrb、wdata 信息寄存。

Tag Compare 将每一路 Cache 中读出的 Tag 和 Request Buffer 寄存下来的 tag 进行比较，生成是否命中的结果。

Data Select 对两路 Cache 中读出的 Data 信息进行选择，得到各种访问操作所需要的结果。对应命中的读操作，首先用地址的 [3:2] 从每一路 Cache 读出的 Data 数据选择出一个字，然后再根据 Cache 命中的结果从两个字中选择出 Load 的结果。对应 Replace 操作，需要根据替换算法决定出的路信息，将读出的 Data 进行选择。

Miss Buffer 用于记录缺失 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。Miss 处理时需要的地址、是否是 Store 指令等信息依然维护在 Request Buffer 中。

LFSR 是线性反馈移位寄存器，采用伪随机替换算法，LFSR 会作为伪随机数源。

Write Buffer 是在 Hit Wire 时启动的，它会寄存 Store 要写入的 Index、路号、写使能和写数据，然后使用寄存后的值真正写入到 Cache 中。

2. 功能描述

Request Buffer 的代码设计如下，以 op 信号为例：

```
reg          rb_op;//寄存所有的查找cache 信息
reg  [  7:0] rb_index;
reg  [ 19:0] rb_tag;
reg  [  3:0] rb_offset;
reg  [  3:0] rb_wstrb;
reg  [ 31:0] rb_wdata;
wire         way0_v;//同时读出来相应的v,d,tag,data
wire         way1_v;
wire         way0_d;
wire         way1_d;
wire [ 19:0] way0_tag;
wire [ 19:0] way1_tag;
wire [127:0] way0_data;
```

```
wire [127:0] way1_data;

always @(posedge clk_g) begin
    if (valid && addr_ok) begin
        rb_op     <= op;
        rb_index  <= index;
        rb_tag    <= tag;
        rb_offset <= offset;
        rb_wstrb  <= wstrb;
        rb_wdata  <= wdata;
    end
end
```

Tag Compare 的代码设计如下：

```
wire         way0_hit;
wire         way1_hit;
wire         cache_hit;//判断是否cache命中
wire         conflict;//判断是否有hit write冲突
assign way0_hit = way0_v && (way0_tag == rb_tag);
assign way1_hit = way1_v && (way1_tag == rb_tag);
assign cache_hit = way0_hit || way1_hit;
assign data_ok = (state == `LOOKUP) && cache_hit || (state ==
   `REFILL) && recv_end;
assign conflict = state == `LOOKUP  && rb_op && valid && !op && (
   offset[3:2] == rb_offset[3:2])
               || wstate == `WRITE && valid && !op && (offset[3:2]
                   == wb_offset[3:2]);
```

Data Select 的代码设计如下：

```
wire [ 31:0] way0_load_word;
wire [ 31:0] way1_load_word;
wire [ 31:0] load_res;
wire [127:0] replace_data;

assign way0_load_word = way0_data[rb_offset[3:2]*32 +: 32];
assign way1_load_word = way1_data[rb_offset[3:2]*32 +: 32];
assign load_res = {32{way0_hit}} & way0_load_word
                | {32{way1_hit}} & way1_load_word;
```

Miss Buffer 的代码设计如下：

```
reg          rp_way_v;//记录需要替换的那一路的信息
reg          rp_way_d;
```

```
reg [ 19:0] rp_way_tag;
reg [127:0] rp_way_data;
reg         wr_req_reg;

always @(posedge clk_g) begin
    if ((state == `LOOKUP) && !cache_hit) begin
        rp_way_v    <= rp_way ? way1_v : way0_v;
        rp_way_d    <= rp_way ? way1_d : way0_d;
        rp_way_tag  <= rp_way ? way1_tag : way0_tag;
        rp_way_data <= rp_way ? way1_data : way0_data;
    end
end
always @(posedge clk_g) begin//查看当前dirty位是否有效
    if (!resetn) begin
        wr_req_reg <= 1'b0;
    end
    else if (state == `MISS && wr_rdy && rp_way_d && rp_way_v)
       begin//如果时dirty data
        wr_req_reg <= 1'b1;
    end
    else if (wr_req_reg == 1'b1) begin
        wr_req_reg <= 1'b0;
    end
end
//当有dirty位是需要向AXI总线发出写请求。
assign wr_req = wr_req_reg;
assign wr_type = 3'b100;
assign wr_addr = {rp_way_tag, rb_index, 4'b0};
assign wr_wstrb = 4'b1;
assign wr_data = {8'hff,rp_way_data[119:0]};
```

Write buffer 的代码设计如下:

```
reg          wb_hit_way;
reg  [  7:0] wb_index;
reg  [  3:0] wb_offset;
reg  [  3:0] wb_wstrb;
reg  [ 31:0] wb_wdata;
always @(posedge clk_g) begin
    if ((state == `LOOKUP) && cache_hit && (rb_op == 1'b1)) begin
        wb_hit_way <= way1_hit ? 1'b1 : 1'b0;
        wb_index   <= rb_index;
        wb_offset  <= rb_offset;
```

```
            wb_wstrb    <= rb_wstrb;
            wb_wdata    <= rb_wdata;
        end
    end
```

3 实验过程

3.1 实验流水帐

12 月 17 日 20：00 至 22：00，读讲义，并完成 cache 表的实例化。
12 月 18 日 8：00 至 10：00，15：00 至 21：00，读讲义，进行代码设计。
12 月 20 日 10：00 至 11：00，完成顶层输出信号的设计，完成代码设计。
17：00 至 19：00，改正所有错误，仿真通过并成功上板。

3.2 错误记录

3.2.1 错误 1：没有对 replace 出的数据进行处理

1. 错误现象
未通过仿真，仿真在开始便错误

2. 定位过程
发现在对于 index00 进行写的读的时候和写的时候的值是一样的，理应不该报错，于是去查看仿真文件对于 replace 错误是如何进行判定的，发现在进行对比的时候，顶层文件把读出来的数据的最高位替换成了 8’bff，后来才知道高位的 ff 不会被修改，所以替换出来要保持不变。

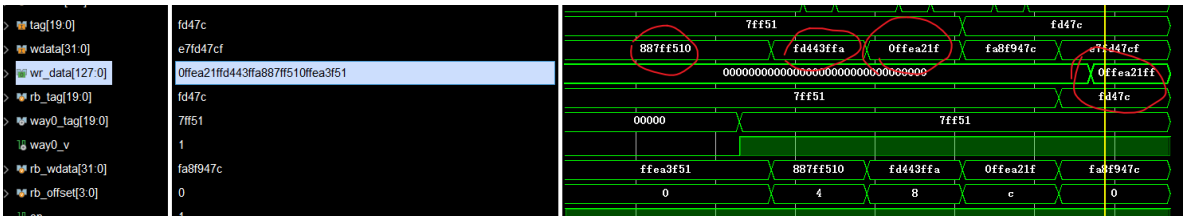

图 2: 上板结果

3. 错误原因
没有先看仿真文件，提前不知道高位的 ff 不会被修改

4. 修正效果

```
    assign wr_data = {8'hff,rp_way_data[119:0]};
```

3.2.2 错误 2：AXI 读入数据未与 wdata 拼合

1. 错误现象

仿真报错，replace 阶段出错。

2. 定位过程

定位到第一次向 bank 写入数据的时刻，此时主状态机处于 refill 阶段，通过 AXI 总线返回的 rdata 直接写入了 bank。再翻看讲义，由于此时为写 cache 缺失之后的处理阶段，返回的数据应该和 wdata 拼合之后写入 bank。进一步查看代码，确实没有考虑这种情况。

3. 错误原因

写 cache 缺失后的处理阶段从 AXI 读入的数据未与 wdata 拼合直接写入 bank。

4. 修正效果

进行数据拼合即可。加一个选择器，选择当前字节应该选择 wdata 还是 rdata.

```
assign rd_way_wdata_bank0[ 7: 0] = (rb_op && rb_offset[3:2] == 2'b00
   && rb_wstrb[0]) ? rb_wdata[ 7: 0] : rd_way_data_bank0[ 7: 0];//拼
   合
assign rd_way_wdata_bank0[15: 8] = (rb_op && rb_offset[3:2] == 2'b00
   && rb_wstrb[1]) ? rb_wdata[15: 8] : rd_way_data_bank0[15: 8];
assign rd_way_wdata_bank0[23:16] = (rb_op && rb_offset[3:2] == 2'b00
   && rb_wstrb[2]) ? rb_wdata[23:16] : rd_way_data_bank0[23:16];
assign rd_way_wdata_bank0[31:24] = (rb_op && rb_offset[3:2] == 2'b00
   && rb_wstrb[3]) ? rb_wdata[31:24] : rd_way_data_bank0[31:24];
```

3.3 实验结果

仿真通过 0xff 个测试点

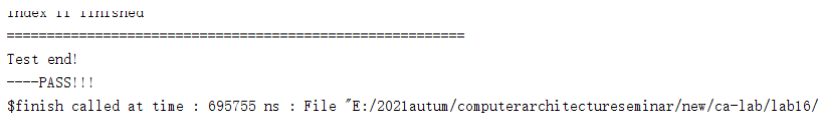

图 3: 上板结果

上板也成功

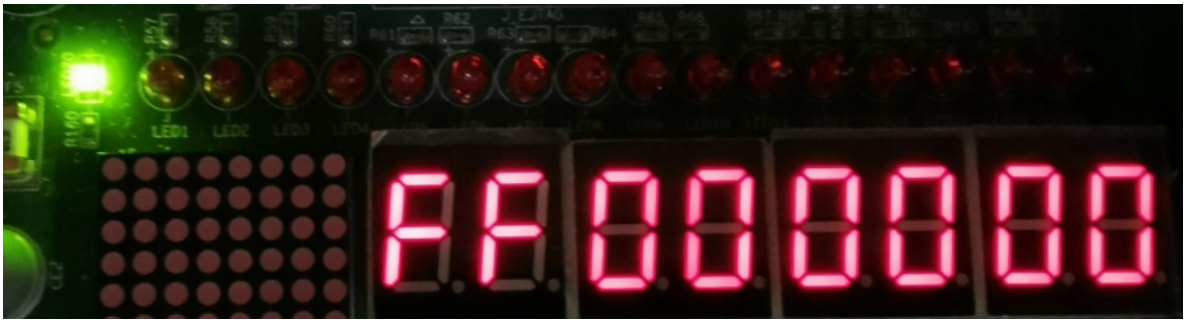

图 4: 上板结果

4 实验总结

通过本次实验，我们理解 Cache 的组织结构和工作机理。也知道了诸如 tagv，bank 在实际中是如何是实现的。也知道了如何处理 cache 和 AXI 总线以及 CPU 的交互。