实验 16 报告

学号	2019K8009929019	2019K8009929026
姓名	桂庭辉	高梓源
箱子号	44	

1 实验任务

自 AXI 总线专题后，我们可以注意到整体性能随着访存延迟的增大而显著下降。在实际设计中，CPU 的运行速度增长速度远快于内存，在存储单元上存储密度与运行频率难以兼得，CPU 与内存频率间的剪刀差导致了更大的访存开销。为了解决这样的问题，在 CPU 与内存间引入多级存储层次。在靠近处理器流水线处使用存储密度小而频率高的电路结构，在远离流水线处使用存储密度高的低速存储。在这一体系中主要的运算逻辑（CPU 流水线中寄存器）与主要存储设备内存间的一层主要是 Cache。现代处理器通常采用三级 Cache 结构，其中通常的组织形式为 L1 Cache 拆分为指令 Cache 与数据 Cache，L2 与 L3 均为单个大数据 Cache。对多核处理器通常 L1、L2 Cache 由每个核心独有，L3 Cache 在若干个核心间共有。

由于上述讨论的 Cache 种类中只有 L1 的指令 Cache 对访存端口只读不写，其他 Cache 均需支持写功能，所以本次实验设计一种支持读写的通用 Cache。忽略 Cache 集成相关的内容，仅关心 Cache 内部逻辑可归纳出如下规格要求：

1. Cache 容量为 8 KB，映射形式采用两路组相联，Cache 行大小为 16 Byte。
2. Cache 采用 Tag 和 Data 同步访问的形式。
3. 替换策略采用伪随机替换算法。
4. 采用阻塞式设计，在发生 Cache Miss 时，阻塞后续访问直至数据填回 Cache。
5. Cache 不采用“关键字优先”技术。
6. 数据 Cache（即 Cache 的写策略）采用写回写分配策略。

2 实验设计

Cache 状态机的设计来自课程讲义，本报告不再赘述，仅将针对读写是否命中的不同情况讨论各状态的具体行为与部分重要逻辑实现。

对于读命中，Cache 自 IDLE 或 LOOKUP 状态转移到 LOOKUP 状态，并在后一拍返回读数据。

对于读缺失，Cache 根据替换策略选择某一路，检查其中对应行的脏位与有效位。若脏位拉高且行有效，则需要将该行数据同步到 RAM，即自 LOOKUP 状态转移至 MISS，将该行读出从总线写端口发出，写请求发出后转移至 REPLACE 状态。若脏位或有效位拉低，则该行数据与 RAM 中相应位置数据相同，无需进行写操作更新 RAM 内容，可直接转移到 REPLACE 态。在 REPLACE 态 Cache 向总线请求相应地址处的 RAM 数据，在握手完成后转移到 REFILL 态，接收总线返回的数据并写入 Cache，当总线返回 CPU 请求处数据时将其返回给 CPU。

对于写命中，Cache 在 LOOKUP 态判断出写命中 `hit_write`，将 Write Buffer 状态机置 WRITE 态，将相应数据写入 Cache。

对于写缺失，Cache 的替换读出流程与读缺失相同，重填写入时需要将写数据写到总线返回数据上再写入 Cache。

命中时主状态机在 IDLE 与 LOOKUP 间的逻辑，Write Buffer 状态机在 IDLE 与 WRITE 间的状态转移逻辑已由课程讲义给出详细的介绍，此处不再讨论。

Cache 中重要的控制与数据信号包括对 CPU 的 `addr/data_ok` 与返回数据，对总线的读写请求与相关信号，主要的存储单元维护包括 D 表、V-Tag 表、各数据 bank。

`addr/data_ok`、`wr/rd_req` 逻辑同样参考讲义实现，对 CPU 返回数据上需区分命中返回 Cache 命中数据与缺失返回总线返回数据，此处的选择信号可以是状态机状态，也可以是读事务中的独有信号，如 `ret_valid`：

```
assign rdata = ret_valid ? ret_data : load_res;
```

对总线的读地址始终为 CPU 请求的地址，写地址中 Tag 字段应来自替换路相应 Cache 行中记录的 Tag 信息，写数据为替换路读出的 Cache 行（V-Tag 表与 data bank 的访问地址后续讨论，保证此处读出内容为 `index_r` 处内容）。

```
assign rd_addr = {tag_r, index_r, offset_r};

assign wr_addr  = {vtag_rdata[replace_way][19:0], index_r, offset_r};
assign wr_data  = {data_bank_rdata[replace_way][3],
                   data_bank_rdata[replace_way][2],
                   data_bank_rdata[replace_way][1],
                   data_bank_rdata[replace_way][0]};
```

D 表的维护在于 Cache 行中数据与相应内存区间的数据不再相同后置 1，重填 Cache 行时需注意此时填入 Cache 行数据是否与内存相同，对于写缺失，CPU 发来的写数据已经修改内存发来的数据，应置 1；对于读缺失，内存读出的数据未被修改，但此前此处脏位可能为 1，需置零保证 D 表的正确维护。所以 D 表的维护发生在两处，一处在于 Write Buffer 写数据 bank，修改 Cache 行需强制置 1，另一处在于重填 Cache 行，需根据请求类型进行更新。

```
always @ (posedge clk_g) begin
    if (rst) begin
        dirty_arr[0] <= 256'b0;
        dirty_arr[1] <= 256'b0;
    end else if (wrbuf_cur_state[`WRBUF_WRITE]) begin
        dirty_arr[wrbuf_way][wrbuf_index] <= 1'b1;
    end else if (ret_valid & ret_last) begin
        dirty_arr[replace_way][index_r] <= op_r;
    end
end
```

V-Tag 表的写更新操作来自重填 Cache 行时，填入的 V 位始终为 1，Tag 字段来自请求缓存，写地址同样来自请求缓存。V-Tag 表的读地址有在接收 CPU 请求时读请求端口的 `index` 处数据，另有生成 `wr_addr` 时读 `index_r` 处数据。所以可以看出 V-Tag 表的访问地址在 IDLE 与 LOOKUP 态（下一拍可能对读数据进行 Tag 比对）为端口输入，其他情况均来自 `index_r`。访问数据 bank 的地址有相同的逻辑，下文不再次讨论。

```
assign vtag_we[0] = ret_valid & ret_last & ~replace_way;
assign vtag_we[1] = ret_valid & ret_last &  replace_way;
assign vtag_wdata[0] = {1'b1, tag_r};
assign vtag_wdata[1] = {1'b1, tag_r};
assign vtag_addr[0] = cur_state[`IDLE] || cur_state[`LOOKUP] ? index : index_r;
assign vtag_addr[1] = cur_state[`IDLE] || cur_state[`LOOKUP] ? index : index_r;
```

数据 bank 的写更新操作类似 D 表，来自写命中时的写操作与缺失时重填。生成数据 bank 所用同步 RAM 时勾选字节写使能后可简化写命中时的字节写逻辑，写缺失重填时由于 `wstrb_r` 作用于 `ret_data` 而非数据 bank 内容，故而需在 `wdata` 生成逻辑中完成该步。

```
generate
    for (i = 0; i < 4; i = i + 1) begin
        assign data_bank_we[0][i] = {4{wrbuf_cur_state[`WRBUF_WRITE] & wrbuf_offset[3:2] == i &
        ↪  ~wrbuf_way}} & wrbuf_wstrb
                                  | {4{ret_valid & ret_cnt == i & ~replace_way}} & 4'hf;
        assign data_bank_we[1][i] = {4{wrbuf_cur_state[`WRBUF_WRITE] & wrbuf_offset[3:2] == i &
        ↪  wrbuf_way}} & wrbuf_wstrb
                                  | {4{ret_valid & ret_cnt == i &  replace_way}} & 4'hf;
        assign data_bank_wdata[0][i] = wrbuf_cur_state[`WRBUF_WRITE] ? wrbuf_wdata :
                                       offset_r[3:2] != i || ~op_r   ? ret_data    :
                                       {wstrb_r[3] ? wdata_r[31:24] : ret_data[31:24],
                                        wstrb_r[2] ? wdata_r[23:16] : ret_data[23:16],
                                        wstrb_r[1] ? wdata_r[15: 8] : ret_data[15: 8],
                                        wstrb_r[0] ? wdata_r[ 7: 0] : ret_data[ 7: 0]};
        assign data_bank_wdata[1][i] = wrbuf_cur_state[`WRBUF_WRITE] ? wrbuf_wdata :
                                       offset_r[3:2] != i || ~op_r   ? ret_data    :
                                       {wstrb_r[3] ? wdata_r[31:24] : ret_data[31:24],
                                        wstrb_r[2] ? wdata_r[23:16] : ret_data[23:16],
                                        wstrb_r[1] ? wdata_r[15: 8] : ret_data[15: 8],
                                        wstrb_r[0] ? wdata_r[ 7: 0] : ret_data[ 7: 0]};
    end
endgenerate
```

3 实验过程

3.1 实验流水账

2021.12.19 18:30 ~ 2021.12.19 22.20 ： 完成 Cache 代码设计并通过测试。

2021.12.20 18:00 ~ 2021.12.20 20:00 ： 完成本实验报告撰写。

3.2 错误记录

本次实验报告不再讨论笔误与简单逻辑错误。

3.2.1 错误 1：data_ok 逻辑错误

3.2.1.1 错误现象

读测试中 cache_top 中用于输入的 counter_i/j 与用于比对的 res_counter_i/j 不匹配，导致比对判定读错误。

3.2.1.2 分析定位过程

注意到上述索引错误后比对对应索引处的标准数据，验证读数据与标准数据一致，故而数据维护上并无问题，可能的问题来自控制信号。考察 cache_top.v 中 counter_i/j 由 addr_ok 触发自增，而 res_counter_i/j 由 data_ok 触发，回溯波形注意到存在一个 addr_ok 后 Cache 返回两次 data_ok。

3.2.1.3 错误原因

课程讲义中给出的 data_ok 逻辑中包括一项 “LOOKUP 状态处理写操作”，未意识到该步包括了写命中与写缺失，而第三条描述的 REFILL 态相应条件拉高 data_ok 是读缺失独有，代码设计时未在该处限定读缺失拉高，导致写缺失处理过程中拉高两次 data_ok。

3.2.1.4 修正效果

在 REFILL 态的 data_ok 分支逻辑上与上 ~op_r，避免写缺失重复发 data_ok。

```
assign data_ok = cur_state[`LOOKUP] & cache_hit
               | cur_state[`LOOKUP] & op_r
               | cur_state[`REFILL] & ret_valid & ret_cnt == offset_r[3:2] & ~op_r;
```

4 实验总结

由于组成原理课程中设计过 Cache，所以 Cache 的逻辑策略、组织形式对我们并未带来较大困难，主要的实验阻力来自状态机理解与总线接口交互。课程讲义给我们提供了许多帮助，但也不能不带脑子做实验写代码。