中国科学院大学
《计算机体系结构(研讨课)》实验报告

姓名 裴晨皓 竹彦博 纪弘璐 学号 2023K8009916003 2023K8009916001 2023K8009916002
专业 计算机科学与技术 实验项目编号 Project 7 实验名称 Cache 设计专题实验

1 实验简介

这次实验中，本组同学按照讲义提示，先实现了独立的 Cache 模块，又向其中加入了非缓存访问的处理逻辑，作为 ICache 和 DCache 分别接入流水线；同时修改了 AXI 转接桥逻辑，使其接口与 Cache 面向 AXI 转接桥的接口相匹配，并加入了突发传输逻辑；此外，还对 MMU 模块进行修改，利用其对存储访问类型进行判定，从而决定 Cache 模块内走可缓存/非缓存的处理逻辑；最后，进一步修改 Cache 和流水线模块，使其支持 CACOP 指令的处理。

2 设计方案介绍

下图是完成本次实验后的处理器结构框图：

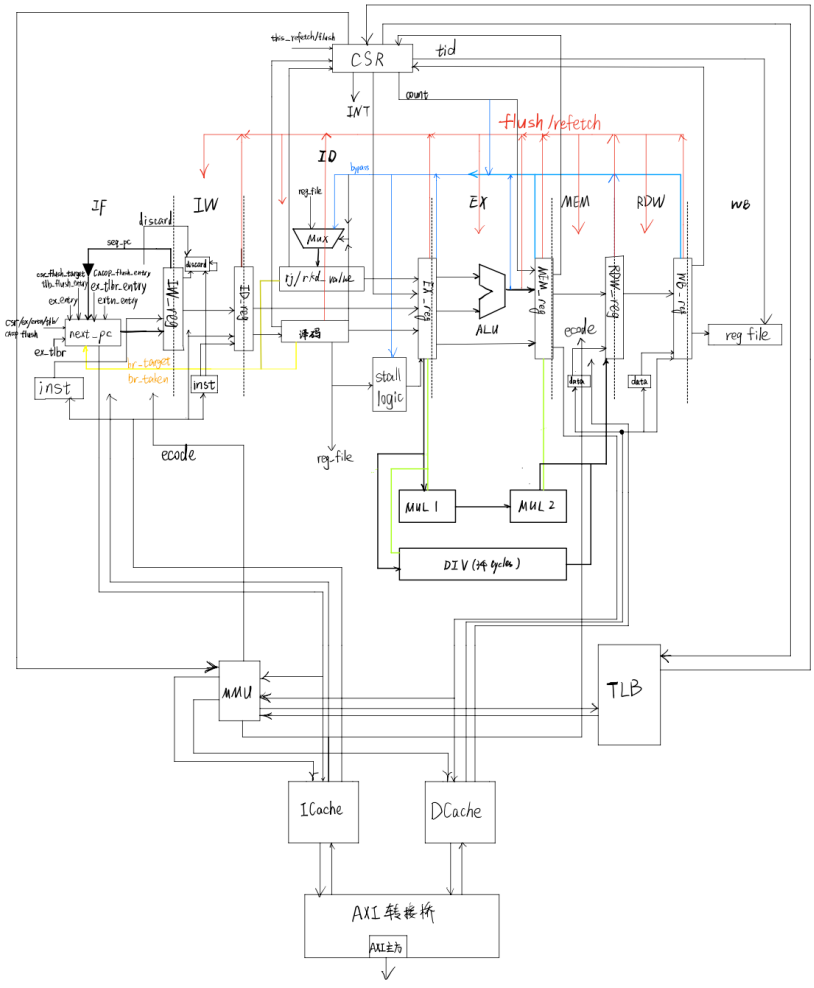

图 1: 处理器结构框图

2.1 总体思路

本次实验新增加了 Cache 模块，上面的结构框图还不够直观，下图直观展现新增 Cache 后各主要模块的协同工作情况：

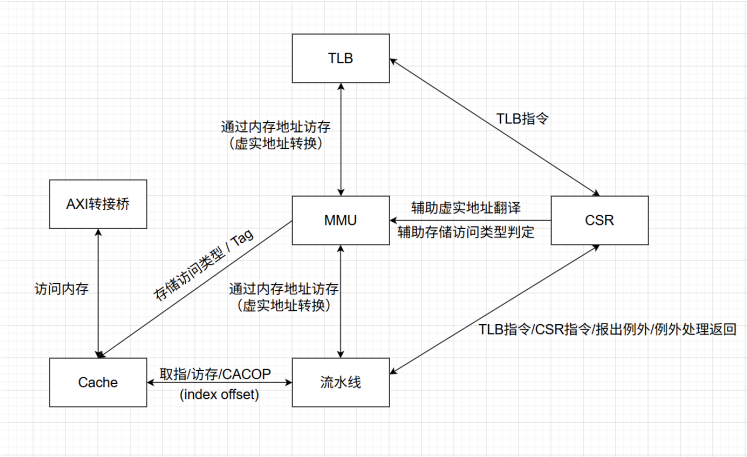

图 2: 协同工作逻辑图

2.2 Cache 模块设计

2.2.1 对 Cache 请求的接收

按照讲义说明，空闲的 IDLE 状态和 LOOKUP 状态都是可以接收流水线发来的请求的状态。如果主状态机处于这两个状态，并且在流水线发来 Cache 请求/CACOP 指令时，只要不存在冲突 (阻塞)，Cache 就可以接收它。

具体的冲突 (阻塞) 情况如下：

```
wire stall =
    (m_current_state == M_LOOKUP) && hit && !cacop_hit_inv_reg && (op_reg == 1'b1) && valid && ({tag, index, offset[3: 2]} == {tag_reg, index_reg, offset_reg[3: 2]}) ||
    (m_current_state == M_LOOKUP) && hit && !cacop_hit_inv_reg && (op_reg == 1'b1) && cacop ||
    (w_current_state == W_WRITE)  && (valid && (op == 1'b0) && ((offset[3: 2] == w_offset_reg[3: 2]) || (index != w_index_reg)) || cacop);
```

图 3: 来自流水线的 Cache 请求-冲突 (阻塞) 情况 (单行代码较长, 故采用截图)

1. 数据冲突：

   如果正在处理一条写命中 Cache 请求：

   - 若新的正常访存请求地址与正在处理的请求地址存在写后读冲突，那么需要阻塞。
   - 若新发来的请求为 CACOP，也需要阻塞。因为 CACOP 指令与命中的 store 指令一样，涉及对 Dirty 位的改写，因此需要保证它们被顺序处理 (顺序改写 Dirty 位)，这里选择在 store 处理完后再让 CACOP“进来”。

2. 结构冲突：

   - 根据讲义提示，若 Write Buffer 状态机正处于 WRITE 状态，说明正在处理对 Cache 的写入，若流水线再发来一个 Load 类的 Cache 访问请求，与正在处理的写请求访问的是同一个 Bank，则由于 Bank 的实现使用的是单端口 RAM，故不能马上接收这一新请求。
   - 还是由于单端口 RAM 的特性，我们的设计中，RAM 读写共用一个地址信号，在 Write 时这一地址信号被设为 Write Buffer 里保存的地址，而非新请求的地址，因此新请求的 RAM 地址与 Write Buffer 里保存的地址不同时，应当阻塞新请求，否则会从错误的 RAM 地址做读取。

进一步地，我们可以得到 Cache 接收新请求的条件，即 addr_ok/cacop_ok 的更新逻辑：

```
assign addr_ok = !cacop && ((m_current_state == M_IDLE && !stall) || ((m_current_state ==
    M_LOOKUP) && hit && !cacop_hit_inv_reg && valid && !stall));
assign cacop_ok = (m_current_state == M_IDLE && !stall) || ((m_current_state == M_LOOKUP) && hit
     && !cacop_hit_inv_reg && cacop && !stall);
```

Listing 1: 对 Cache 请求的接收-addr_ok/cacop_ok

- 在没有阻塞的情况下，若是处在 IDLE 或是有正常访存 hit 的 LOOKUP 阶段，则可以接受请求。
- 在 LOOKUP 阶段有可能存在 hit 的 CACOP 指令，但由于后续状态机还要进入 MISS 等阶段做 Cache 的写回，还要用到 Request Buffer 里保存的信息，故这时不能接受新请求。
- 在我们的设计中，为了保障性能，减少 CACOP 造成的指令重取开销，所以在 CACOP 指令访问 icache 和 IF 级访问 icache(取指) 同时发生时，优先相应 CACOP，拉高 cacop_ok，拉低 addr_ok。

接收请求的时候，更新 Request Buffer：

```
else if(((m_current_state == M_IDLE) && cacop && !stall) || ((m_current_state == M_LOOKUP) &&
    hit && !cacop_hit_inv_reg && cacop && !stall)) begin
    op_reg <= 1'b0;
    cached_reg <= 1'b1;
    tag_reg <= cacop_tag;
    index_reg <= cacop_index;
    offset_reg <= cacop_offset;
    wstrb_reg <= 4'b0;
    wdata_reg <= 32'b0;
    cacop_st_tag_reg <= cacop_st_tag;
    cacop_idx_inv_reg <= cacop_idx_inv;
    cacop_hit_inv_reg <= cacop_hit_inv;
end
else if(((m_current_state == M_IDLE) && valid && !stall) || ((m_current_state == M_LOOKUP) &&
    hit && !cacop_hit_inv_reg && valid && !stall)) begin
    op_reg <= op;
    cached_reg <= cached;
    tag_reg <= tag;
    index_reg <= index;
    offset_reg <= offset;
    wstrb_reg <= wstrb;
    wdata_reg <= wdata;
    cacop_st_tag_reg <= 1'b0;
    cacop_idx_inv_reg <= 1'b0;
    cacop_hit_inv_reg <= 1'b0;
end
```

Listing 2: 对 Cache 请求的接收-Request Buffer 更新 (时序逻辑块代码片段节选)

若是 CACOP 指令，则向 Request Buffer 存入 CACOP 指令的具体类型、index、tag 等，并把存储访问类型置为可缓存 (因为 CACOP 访问的就是 Cache 本身)；若是正常访存，则把正常访存对应的 tag、index 等信息保存下来，并且根据实际情况设定可缓存/非缓存。

2.2.2 正常访存 Cache 命中情况的处理

主状态机在 LOOKUP 状态对命中进行判定，若这一周期没有新的请求或是请求被阻塞，则回到 IDLE 空闲状态；若有未被阻塞的新请求，则下一拍仍在 LOOKUP，对新请求进行处理。

在可缓存时 (Cacop 采用查询索引时也复用此逻辑)，针对每一路 Cache，对同一 index，若请求的 tag 与 Cache 中存储的 tag 相同，且 cache 行有效，那么就为命中：

```
wire hit_way_1 = (tagv1_rdata[20: 1] == tag_reg && tagv1_rdata[0]) && (cached_reg ||
    cacop_hit_inv_reg);
wire hit = hit_way_0 || hit_way_1;
```

Listing 3: 命中判定

读命中比较简单，根据命中的是哪一路，选择相应的数据传给流水线即可。

若是写命中，要把写相关信息以及命中哪一路的信息保存在 Write Buffer，并且还要把命中的 Bank 内相应 index 的位置上原来的“字”暂存下来，然后与流水线发来的一个字大小的写数据通过 wstrb 掩码进行合并、拼接，得到最终写回 Cache 的数据：

```
wire [31: 0] hit_mask = {{8{w_wstrb_reg[3]}}, {8{w_wstrb_reg[2]}}, {8{w_wstrb_reg[1]}}, {8{
    w_wstrb_reg[0]}}};
wire [31: 0] hit_wdata = (w_wdata_reg & hit_mask) | (w_prev_data & ~hit_mask);
```

Listing 4: 写命中情况下最终写回 Cache 的数据

在 Write Buffer 状态机处于 WRITE 状态时，即可将数据写回 Cache，并更新 Dirty 位。

2.2.3 正常访存 Cache 未命中情况的处理

发现未命中后，主状态机会由 LOOKUP 状态转到 MISS 状态，做 Cache 替换。被替换的 Cache 行若 Dirty 位为 1，则先拉高 wr_req 请求把数据写回内存，等待 AXI 转接桥能够接收后 (wr_rdy 拉高)，跳转到 REPLACE。

在 REPLACE 拉高 rd_req，请求把新行从内存通过突发传输读出来，待 AXI 转接桥能够接收请求后 (rd_rdy 拉高)，跳转到 REFILL 状态，接收内存突发传回的数据，并存入 Cache 的 data_ram 里。

不难想到，内存传回的 16B 位于不同的 Bank 里，故此处设立一计数器，每接收到一个字，计数器就增 1，用于辅助把数据写入不同的 Bank(传完后，last 拉高时要重置，以便下次传输)：

```
always @(posedge clk) begin
    if (rst) begin
        read_cnt <= 2'b0;
    end
    else if (ret_last == 1'b1) begin
        read_cnt <= 2'b0;
    end
    else if (ret_valid) begin
        read_cnt <= read_cnt + 2'b1;
    end
end
```

Listing 5: 突发读-计数器

由于传回的第 1 个字写入被替换路的第 0 个 Bank，第 2 个字写入第 1 个 Bank，以此类推，因此根据计数器设计如下信号，生成 Bank 选择的独热码——第几个 Bank 被选中时，对应位为 1，其余位为 0：

```
wire [3: 0] data0_wbank_sel = (m_current_state == M_REFILL) ? (4'b1 << read_cnt) : (4'b1 <<
    w_offset_reg[3: 2]);
```

Listing 6: Bank 选择信号

(上面的 w_offset_reg 为写命中时的 Bank 选择，类似地，也是用其生成独热码。)

当命中 (选中) 第 0 号 Bank 时，如果第 0 路写控制信号拉高 (即 REFILL 状态下第 0 路被替换)，那么就把这一个 Bank 的字节写使能拉高，否则置全 0，其它 Bank 与此类似：

```
assign data0_bank0_we = ((data0_wbank_sel == 4'h1) && data0_we) ? 4'b1111 : 4'b0;
```

Listing 7: 写 Bank 选择信号转换为写使能信号

上面提到的第 0 路写控制信号用于控制“这一拍是否要对第 0 路做写操作”，具体如下：

```
wire data0_we = !cacop_reg && cached_reg
            && (((m_current_state == M_REFILL) && (replace_way == 1'b0) && ret_valid)
             || ((w_current_state == W_WRITE) && (w_way_reg == 1'b0) && w_we_reg));
```

Listing 8: 第 0 路写控制信号

data0_we 拉高的条件为：在非缓存情况下，在 REFILL 状态重填第 0 路，且数据从 AXI 转接桥发回 (ret_valid)；在写命中第 0 路的情况下，Write Buffer 处于 Write 状态。

还要注意的是，store 操作未必是对整个字的修改，所以在写未命中情况的 REFILL 状态下，重填被 store 修改的那个字时，要特别处理：

```
wire [31: 0] refill_mask = {{8{wstrb_reg[3]}}, {8{wstrb_reg[2]}}, {8{wstrb_reg[1]}}, {8{
    wstrb_reg[0]}}};
wire [31: 0] refill_wdata = (ret_data & ~refill_mask) | (wdata_reg & refill_mask);
```

Listing 9: 写未命中-数据拼接

首先根据 wstrb 生成按 bit 的掩码 mask，然后用掩码把 AXI 转接桥传回的完整字的相应部分改成 store 指令写入的值即可。

这时，结合前面几种情况，我们可以得出向 Cache 中写入的数据的选择逻辑：

```
assign data0_wdata = (w_current_state == W_WRITE) ? hit_wdata :
                     !op_reg ? ret_data :
                     (read_cnt == offset_reg[3: 2]) ? refill_wdata : ret_data;
```

Listing 10: 写入 Cache 的数据选择逻辑

这三个条件依次对应于：写命中、读未命中 (需要写入 AXI 转接桥发回的 ret_data)、写未命中 (如果刚好是 CPU 发来的 store 指令修改的那个字，要向 Cache 写入拼接后的内容 refill_wdata，其他情况直接填入 AXI 转接桥返回的字)

此外，这里还要注意，发给内存的写数据是整行一次发回的，而从内存读回的数据是逐字接收的。

2.2.4 非缓存情况的处理

非缓存的情况下，虽然访存请求会经过 Cache 模块，但并不会与 Cache 内存储的信息进行交互。这种情况状态机的转换路径为：主状态机在 LOOKUP 接收请求后直接进入 MISS，然后分读写两种情况：

1. 若是读，则下一拍直接进入 REPLACE 状态向 AXI 转接桥发读请求，握手成功后进入 REDILL，收到数据后回到 IDLE。
2. 若是写，则向 AXI 转接桥发送写请求，握手成功后进入 REPLACE，等待 AXI 转接桥发来数据写入成功信号 data_ok(wr_complete) 后，回到 IDLE。

这里要注意的是，非缓存一定要等数据真正写入内存，才能回 IDLE 接收新的请求，在此之前的新请求都需要被阻塞。从而保证了使用非缓存访问 IO 外设的正确性。

这时，我们就能写出完整的发给流水线的 data_ok 信号逻辑：

```
assign data_ok = !cacop_reg && (((m_current_state == M_LOOKUP) && hit) ||
        (m_current_state == M_REFILL) && !cached_reg && (op_reg == 1'b0) && ret_valid &&
            ret_last ||
        (m_current_state == M_REPLACE) && !cached_reg && (op_reg == 1'b1) && wr_complete ||
        (m_current_state == M_REFILL) && cached_reg && ret_valid && (read_cnt == offset_reg[3:
            2]));
```

Listing 11: data_ok 信号逻辑

上述代码中：

1. 第一行为缓存命中，直接发 data_ok；第 4 行为未命中重填，访存指令访问的那一个字的数据从 AXI 转接桥发回 (根据计数器和 offset 判断)，就可以认为获得了 load 指令需要的字 (的部分)/store 指令修改的字，于是也可以发 data_ok。

2. 第二、三行是非缓存情况，与前文描述一致，load 指令拿到数据/store 指令真正完成写入 (wr_complete)，就发出 data_ok。

此外还要注意，可缓存情况下向内存发送的请求地址是按 Cache 行对齐的，可能与流水线发给 Cache 的地址不同；而非缓存情况下则是按字对齐 (4 字节)，与流水线发来的地址相同——二者要做区分，以读请求地址为例：

```
assign rd_addr = cached_reg ? {tag_reg, index_reg, 4'b0000} : {tag_reg, index_reg, offset_reg};
```

Listing 12: 内存读请求地址

2.2.5 CACOP 指令的添加

CACOP 指令共有 3 种，有 2 种采用的都是地址直接索引，还有 1 种使用查询索引，使用查询索引的这种与普通 load 指令类似，需要对命中与否进行判断。这 3 种指令的共同点是，如果操作的相应 Cache 行是脏的，需要先将其写回内存 (手册未说明 Store Tag 这种情况是否要写回脏行，我们的设计中采取了写回的方式处理)。

在状态转移方面，使用查询索引的方式若未命中，则没有任何后续处理，直接回到 IDLE；其他情况则一律进入 MISS。若处理的是脏行，则复用替换时写回脏行的逻辑，需要拉高 wr_req，向 AXI 转接桥发送写请求。待 AXI 接口发回的 wr_rdy 拉高，能够接收写请求后进入 REPLACE 状态，在 REPLACE 状态等待数据完成写入 (wr_complete 拉高) 后，在 REFILL 状态对 tag/v 位进行处理，最后回到 IDLE。

CACOP 的写回逻辑复用了未命中做替换时的逻辑，所以相当于也“替换”走了某一行，所以复用原有的 replace_way 逻辑 (在原有基础上做添加)：

```
else if ((m_current_state == M_LOOKUP) && (!hit || hit && cacop_hit_inv_reg)) begin
    if(cacop_st_tag_reg || cacop_idx_inv_reg) begin
        replace_way <= offset_reg[0];
    end
    else if(cacop_hit_inv_reg) begin
        replace_way <= hit_way_0 ? 1'b0 : 1'b1;
    end
    else begin
        replace_way <= rand_way;
    end
end
```

Listing 13: “替换 (写回内存)”路选择

若是采用地址直接索引的两类 CACOP，则根据偏移量 offset 的最低位选择操作哪一路，所以写回内存的与它是同一路；若是采用的查询索引方式，则操作/写回的是命中的那一路。

根据前面的描述：CACOP 要根据具体操作的那一路，产生 tagv0/1 的写使能，清零 tag/v 位——于是得到下面的逻辑：

```
assign tagv0_we
= (cacop_st_tag_reg || cacop_idx_inv_reg) ? ((m_current_state == M_REFILL) && !offset_reg[0]) :
  (cacop_hit_inv_reg) ? ((m_current_state == M_REFILL) && (replace_way == 1'b0)) :
  (cached_reg && (replace_way == 1'b0) && ret_last); //可缓存，从内存获取完重填数据
assign tagv0_wdata = cacop_reg ? 32'd0 : {tag_reg, 1'b1};
```

Listing 14: tagv 写逻辑 (以第 0 路为例)

类似地，相应 dirty 位也要做修改 (归零)，此处不再赘述。

此外，综合前面的所有情况，我们可以得到发给内存的 wstrb：

```
assign wr_wstrb = (cached_reg || cacop_reg) ? 4'b1111 : wstrb_reg;
```

Listing 15: 发给内存的 wstrb

只有非缓存情况下才要发送 Request Buffer 里保存的值；其余情况都是整行写回，即每一个字都要完整写回，所以 wstrb 置为全 1。

2.3 对类 SRAM-AXI 转接桥的修改

这部分修改较为简单，只需要让 AXI 转接桥支持突发传输；并且修改原来的类 SRAM 接口，使转接桥接口与 Cache 模块面向 AXI 转接桥的接口相匹配，然后使转接桥顶层仲裁、分发逻辑与修改后的接口相匹配即可。

2.3.1 对顶层模块的修改

原先对读写请求的区分，要先通过 sram_req 判断有无请求，再根据 sram_wr 区分是读还是写，而现在 Cache 方面会直接发来明确的读写请求，所以要对部分逻辑做相应更改。

以 ar_id 为例：

```
assign ar_id = (~sram_wr_2 && sram_req_2) ? 2'b10 : (~sram_wr_1 && sram_req_1) ? 2'b01 : 2'b00;
```

Listing 16: ar_id-原

```
assign ar_id = rd_req_d ? 2'b10 : rd_req_i ? 2'b01 : 2'b00;
```

Listing 17: ar_id-现

(ar_id 为顶层用于仲裁的独热码，10 表示处理访存级 dcache 的请求，01 表示处理取值级 icache 的请求，00 表示无请求。并且，两个流水级请求同时到来时，为保障性能，先处理访存级 dcache 的请求。)

为了支持突发，要引入 len 信号，表示突发传输的长度：

```
assign ar_len = ar_id[1] ? ((rd_type_d == 3'b100) ? 8'd3 : 8'd0) : ((rd_type_i == 3'b100) ? 8'd3
    : 8'd0);
```

Listing 18: ar_len 信号

这行代码的 type 信号由 Cache 发来，是这次请求传输的字节数以 2 为底的对数 (3'b100 表示 16 字节，3'b010 表示 4 字节)。所以最终长度的生成逻辑为：先跟据回应/处理请求的独热码 ar_id 做仲裁，对取指 icache/访存级 dcache 进行选择，然后根据相应 type，把 len 设为 3(传输 4 次，单次传 4 字节，共 16 字节) 或 0(传输 1 次，单次传 4 字节，共 4 字节)。aw_len 与此类似，不过由于取指级不会发写请求所以省去了仲裁，更加简单，不再赘述。

此外，由于 Cache 发来的写请求是一次性发给转接桥所有数据，所以要把输入的 wdata 扩展到 128 位以应对整行写回；而 Cache 发来的读请求，在接收数据时，都是逐字接收，所以不需要修改 rdata 的宽度。进一步考虑：

- 逐字接收时，AXI 要给 Cache 发送 last 信号，表示当前传输的是最后一个字；

```
    assign ret_last_i = r_id[0] & r_last;
    assign ret_last_d = r_id[1] & r_last;
```

Listing 19: last 信号

根据仲裁独热码，把读响应通道的 last 发给 icache/dcache 即可。

- 一次性写回 16B 时，AXI 顶层模块会把它完整的转交给写请求/数据通道模块，不在顶层进行处理。那么写操作 last 信号也就不用在顶层处理了，只要写请求/数据通道模块内在完整传输的最后一次写时拉高 last，用顶层的 AXI 接口传给内存即可。此外，显然这一信号也不必传给 Cache。

2.3.2 写请求/数据/响应通道模块

此处需要增加设计，便于实现写突发传输。具体地，首先要加一个计数器，记录已经传输了多少个字：

```
always @(posedge clk) begin
    if (!resetn) begin
        counter <= 8'd0;
    end
    else if (wready && wvalid && wlast) begin
        counter <= 8'd0;
    end
    else if (wready && wvalid) begin
        counter <= counter + 8'd1;
    end
end
```

Listing 20: 写通道-计数器

每完成一次写数据握手，计数器就加 1；当 last 拉高时说明已经是最后一次传输，于是把计数器清零，以便下次传输。

一次传输的数据可能长达 16B，但每次传输只能发 4B，所以要利用 counter 计算偏移量，选择当前要发给内存的数据：

```
assign wdata = data_reg[counter * 32 +: 32];
```

Listing 21: wdata 信号

在计数器数值（已完成传输的数量）与本次写操作的 len 相等时，说明本次写操作的最后一个写数据已经发出，于是拉高 last 信号：

```
assign wlast = (counter == len_reg);
```

Listing 22: wlast 信号

此外，还要对状态机的转移条件进行修改，由于突发传输是一次请求握手，多次数据握手，所以：原先以“写数据通道握手成功”为条件进行转移的路径，要把转移条件改为“写数据通道数据传输完毕”，即向其中加入 wlast 信号：

```
BUSY: begin
    if (awready && awvalid && wready && wvalid && wlast) begin
        next_state = IDLE;
    end
    else if (awready && awready) begin
        next_state = AW_FIRE;
```

```
    end
    else if (wready && wvalid && wlast) begin
        next_state = W_FIRE;
    end
    else begin
        next_state = BUSY;
    end
end
AW_FIRE: begin
    if (wready && wvalid && wlast) begin
        next_state = IDLE;
    end
    else begin
        next_state = AW_FIRE;
    end
end
```

Listing 23: 修改后的状态转移逻辑 (局部)

2.3.3 读请求/响应通道模块

这部分修改较为简单：

- 读请求通道接收并保存请求信息时，要额外保存 len 信号，从而作为 arlen 发给内存；
- 读响应通道接收内存发来的读数据时，要额外保存 last 信号，从而传给 Cache 模块，表示当前传输的是否为最后一个字。

2.4 对流水线的修改-加入 CACOP 指令支持

由于使用查询索引方式的那一类 CACOP 指令的 VA 可被当做一条普通 load 指令访问 Cache，所以我们的 CACOP 与 load 访存指令放在同一级实现 (MEM 级)。此外，由于操作指令 Cache 的 CACOP 指令和取指有特权资源相关冲突，所以要仿照上一章，新增一种重取机制。

CACOP 指令操作类型有 3 种，操作的对象有 icache 和 dcache 这 2 种，共计六种情况，每一种情况有单独的信号发给 Cache，拉高即表明发送相应请求：

```
assign cacop_st_tag_i  = in_valid && cacop && !cacop_i_fire && (inst_4_0[2:0] == 3'b000) && (
    inst_4_0[4:3] == 2'b00) && !this_flush && !this_tlb_flush && !this_cacop_flush;
```

Listing 24: CACOP 向 Cache 发送的请求 (以对 icache 的 store tag 类型为例)

以对 icache 的 store tag 类型为例：若当前 MEM 级为 CACOP 指令，且根据指令中的 code(inst_4_0) 判断为操作 icache 的 store tag 类指令，并且请求没有被接收 (没有握手成功，cacop_i_fire 为 0)，且当前指令后续不需要被冲刷，那么就持续拉高 cacop_st_tag_i，向 Cache 发请求。

在 CACOP 指令收到 Cache 发回的 cacop_ok 信号后，拉高 cacop_i_fire，表示针对 icache 的这条指令握手成功：

```
always @(posedge clk) begin
    if(rst) begin
        cacop_i_fire <= 1'b0;
    end
    else if(in_valid && ready_go && out_ready) begin
        cacop_i_fire <= 1'b0;
    end
```

```
    else if(cacop_req_i && cacop_ok_i) begin
        cacop_i_fire <= 1'b1;
    end
end
```

Listing 25: cacop_i_fire 信号

(这条指令从 MEM 级流出时，cacop_i_fire 要清零，以便下次使用。)

由于 CACOP 指令也需要握手，所以没握手成功时要被阻塞在 MEM 级：

```
assign ready_go = ...
!(cacop && (inst_4_0[2:0] == 3'b000) && !(|mmu_ecode_d) && !(cacop_req_i && cacop_ok_i ||
    cacop_i_fire)) &&
!(cacop && (inst_4_0[2:0] == 3'b001) && !(|mmu_ecode_d) && !(cacop_req_d && cacop_ok_d ||
    cacop_d_fire));
```

Listing 26: ready_go 信号修改 (省略原有内容)

即：如果不是“既没发生 TLB 异常，又没握手成功”CACOP 指令，才能继续流动。

由于采用查询索引方式的那类 CACOP 指令（cacop_hit_inv_i/d）需要进行虚实地址翻译，可能出现 TLB 异常，所以要把它加到异常传递 (检测) 逻辑中：

```
else if (in_valid && ready_go && out_ready) begin
    has_exception_out <= has_exception || ((|mmu_ecode_d) & (res_from_mem || mem_we ||
        cacop_hit_inv_i || cacop_hit_inv_d));
end
```

Listing 27: 修改后的异常传递逻辑

关于 CACOP 指令造成的的重取，基本完全仿照之前 TLB 指令导致重取的机制实现。由于 MEM 级实现 CACOP，所以其流入下一级（RDW）后进行“提交”，从而冲刷前面的流水级，并把取指 PC 设为 CACOP 指令下一条顺序指令的 PC(PC 加 4)。具体细节不再赘述。

2.5 MMU 模块对存储访问类型的判定

以取指级的情况为例：

```
assign mat_i = (crmd_da_value && !crmd_pg_value) ? crmd_datf_value :
               (inst_sram_vaddr[31: 29] == dmw0_vseg_value && dmw0_plv_cond) ? dmw0_mat_value :
               (inst_sram_vaddr[31: 29] == dmw1_vseg_value && dmw1_plv_cond) ? dmw1_mat_value :
                tlb_s0_mat;
```

Listing 28: mat_i 信号生成逻辑

(mat 为 0 时表示非缓存，1 表示可缓存。)

- 直接地址翻译模式下，MAT 由 CSR.CRMD 的 DATM 域决定；
- 直接映射地址翻译模式下，MAT 由所命中的直接映射窗口中的 MAT 配置信息 (dmw0/1_mat_value) 决定；
- 页表映射地址翻译模式下，MAT 由虚实地址转换所用页表项中的 MAT 域配置信息决定。

3 Debug 记录

3.1 忘记接入存储访问类型信号导致向外设发起突发请求

在接入 dcache 进行调试时，发现如下图所示波形：

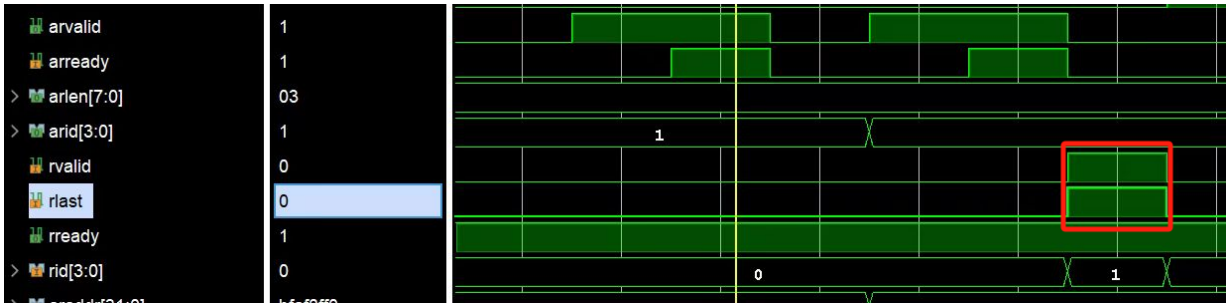

图 4: 忘记接入存储访问类型信号导致向外设发起突发请求-错误波形

上图中光标所在位置处，一次读请求握手成功，arlen 为 3，arid 为 1，是一次 dcache 发起的突发读请求 (读 4 次，每次 4B)。然而，在红框位置处，收到第一个读数据时 (rvalid 拉高)，rlast 居然也拉高了，相当于后续 3 个 4B 没有传回就结束了，不禁令人困惑。

后续检查发现，我们忘记接入存储访问类型的信号，导致所有访存请求都被当做了可缓存访问。上图波形中原本应为非缓存的情况，恰巧 arlen 为 3，导致误企图向外设发起突发请求，而外设并不支持突发传输，故只传回了第一个 4B 数据，后续数据没有传回。

正确接入存储访问类型信号后，问题得以解决。

3.2 两条连续写命中指令导致同一 Bank 同一 Cache 行的“写后读”

对于一条写命中指令，其第 0 个周期会发送请求，第 1 个周期会得到命中结果，并读出相应 Bank、相应 Cache 行的原有数据，第 2 个周期会得出写入值与原值根据 wstrb 拼接后的值，在第 2 个周期进入第 3 个周期的时钟上升沿写回 Cache。最初的设计中，连续两条针对同一 Bank、同一 Cache 行的写命中指令出现下图所示情况：

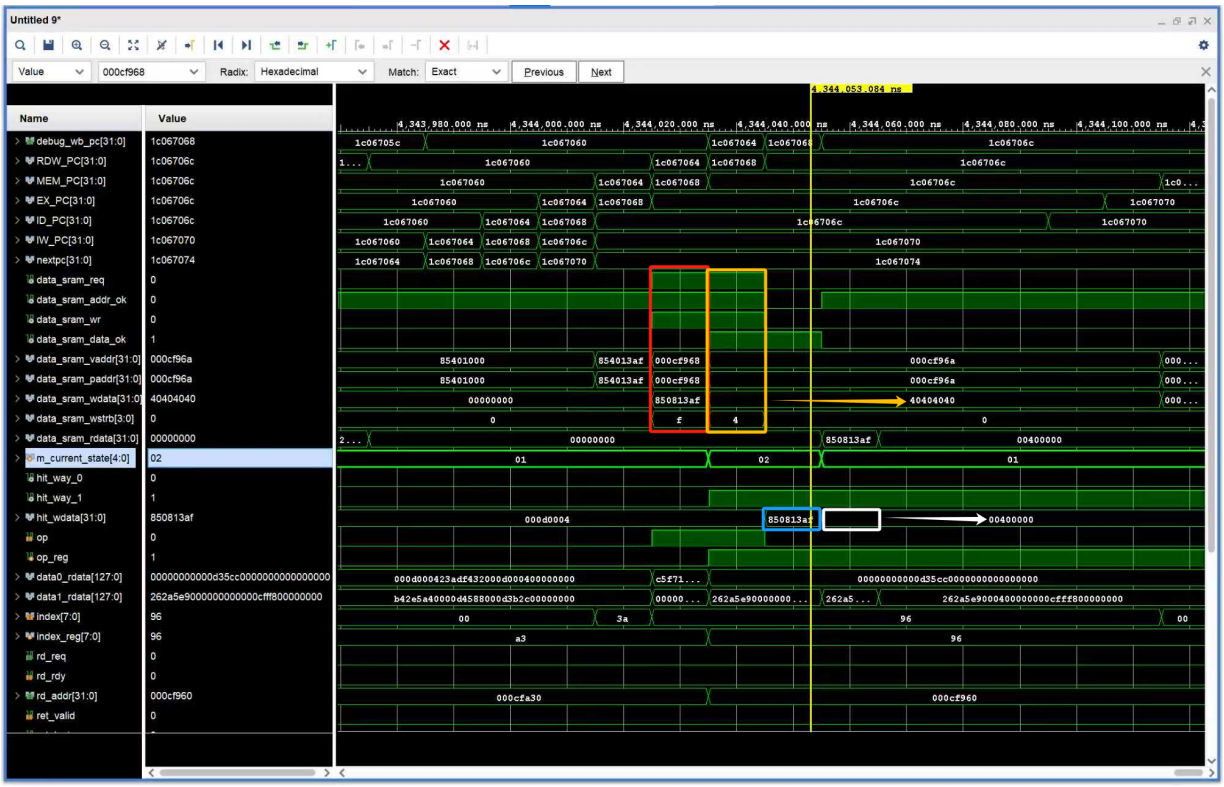

图 5: 两条连续写命中指令导致同一 Bank 同一 Cache 行的“写后读”-错误波形

红框位置是第 1 条写命中指令发来的请求，请求写入 850813af，wstrb 为 1111；黄框位置是第 2 条写命中指令发来的请求，请求写入 40404040，wstrb 为 0100，即只把第 2 字节拼接到 Cache 内原来存储的字里，两条指令命中的是同一 Bank、同一 Cache 行——按照正常时序，最终第二条指令写回 Cache 的字应为 854013af。

然而，在蓝框所示位置处，第 1 条指令才刚刚得到其应当写入 Cache 的值，而此时第 2 条指令也已经读出了命中 Bank 的数据，由于第 1 条指令要在进入下一周期的上升沿才写回 Cache，所以第 2 条指令相当于读出了 Cache 的旧值，出现了“写后读”。毫不意外地，在下个周期，第 2 条指令用 Cache 的旧值与自己写入的字节进行拼接，得出了要写回 Cache 的 00400000，而非正确的 854013af。

最终解决方式为引入阻塞，从而避免这种表面“连续写”，本质暗含“写后读”的情况出现。

3.3 Cache 阻塞时忘记拉低 addr_ok

我们设计的 stall 信号是用于阻塞 Cache 的，即拉高 stall 时不应接收新请求，所以发向流水线的 addr_ok 信号也应当拉低，而最初我们忘了用 stall 信号控制 addr_ok，导致下图情况：

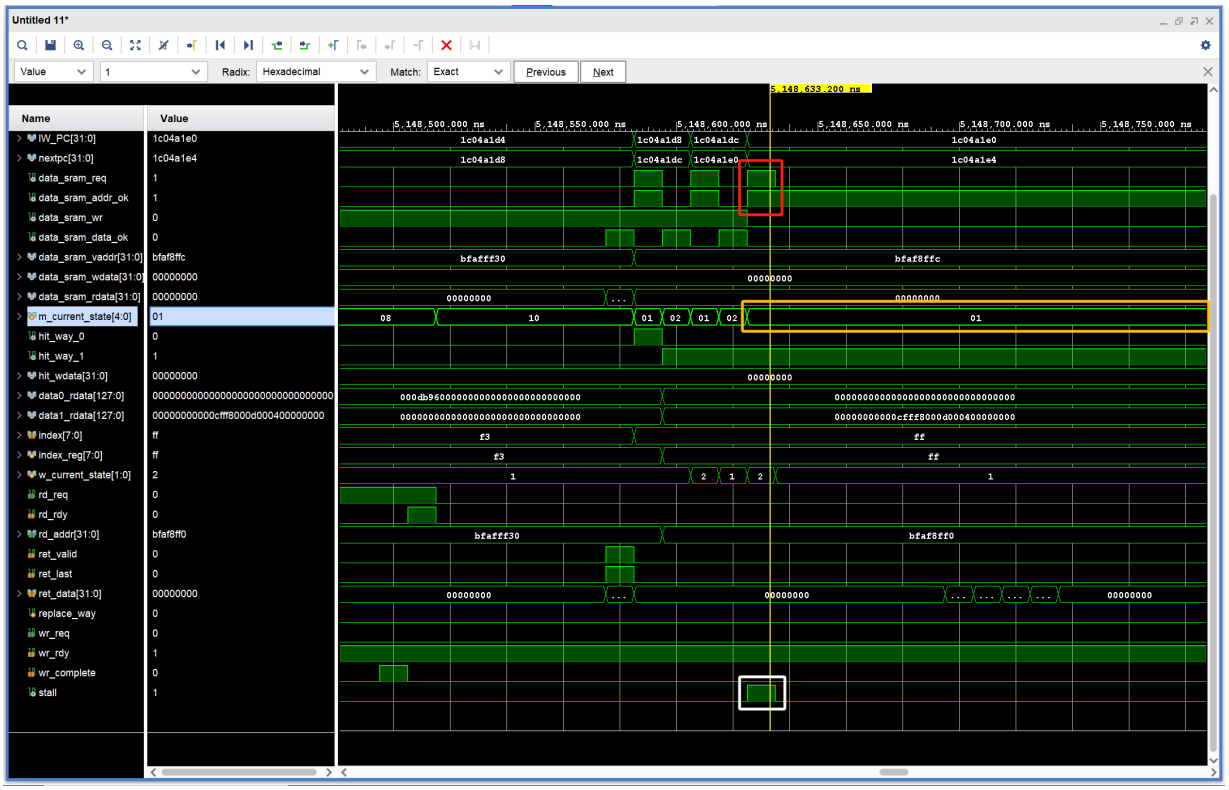

图 6: Cache 阻塞时忘记拉低 addr_ok-错误波形

在白框所示位置处，stall 拉高，而同一周期内，红框位置处却拉高了 addr_ok，与流水线发来的请求“误握手”成功。导致流水线方认为请求已被接收，只需要等待后续 data_ok，而 Cache 方却并不认为自己接收了这次请求，自然也不会给流水线发 data_ok 信号。由于流水线一直等不到 data_ok，后续指令也就都被阻塞，Cache 也收不到新请求，整个系统卡死（如黄框所示，Cache 的状态恒处于 IDLE；图片最上方的 PC 值也不再改变）。

解决方法十分简单，给原有的 addr_ok 信号“与”上“!stall”，限制 stall 拉高时对请求的接受即可。

3.4 突发传输请求地址忘记按 Cache 行对齐

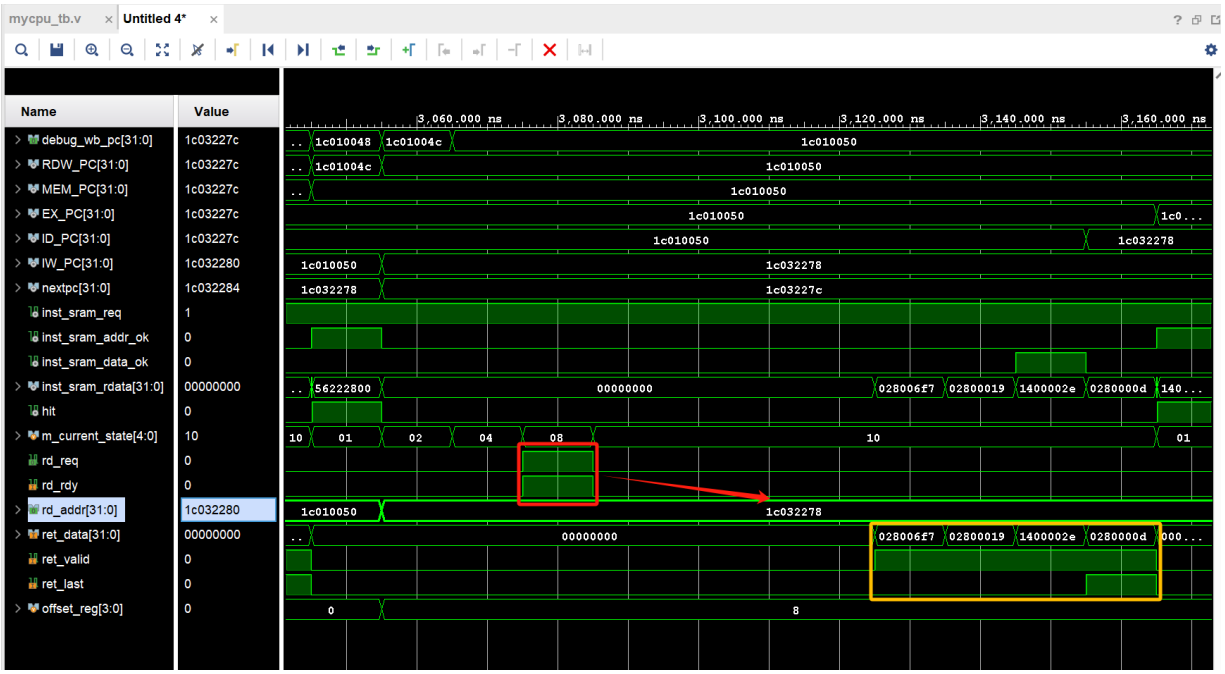

图 7: 突发传输请求地址忘记按 Cache 行对齐-错误波形

如上图所示，Cache 向内存发读请求时，若是可缓存情况下做突发传输，应该把流水线发来的地址按 Cache 行对齐再发给内存；而非缓存情况直接使用流水线发来的地址即可。在加入非缓存处理逻辑时，我们误把原有的突发传输请求地址先按 Cache 行对齐再发送的逻辑，直接改成了发送流水线传来的地址，进而在未按行对齐地址发起突发传输，取回了错误的一行数据。

正确做法是使用选择器将两种请求地址分开：

```
assign rd_addr = cached_reg ? {tag_reg, index_reg, 4'b0000} : {tag_reg, index_reg, offset_reg};
```

Listing 29: 发给内存的读请求地址

4 合作说明

本实验由本组成员共同合作完成，组内同学同等贡献。