

中国科学院大学

《计算机体系结构基础(研讨课)》实验报告

姓名 艾华春,李霄宇,王敬华

学号 2022K8009916011,2022K8009929029,2022K8009925009

实验项目编号 7 实验名称 高速缓存设计

一、 逻辑电路结构与仿真波形的截图及说明

• Cache 模块设计。

1. Cache 的逻辑组织结构

Cache 采用两路组相连的方式,每一路包含相同的 tag,v 表, d 表, 4 个 data_bank 表。所有的表同一时间只能接受一个读或写操作。

如下图所示:

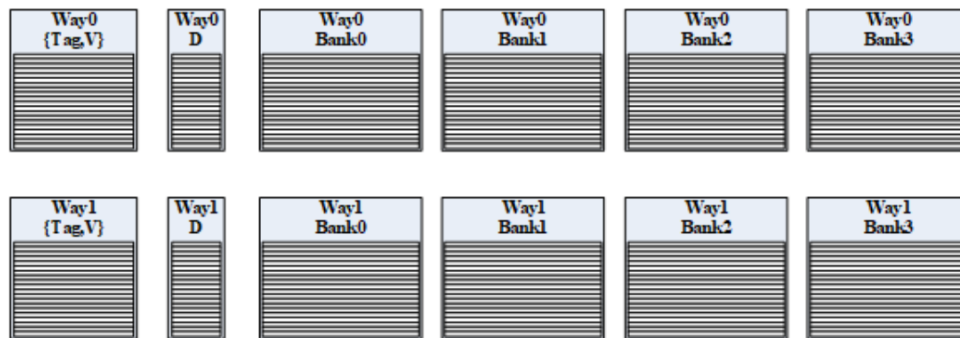


图 1: Cache 逻辑组织结构

(a) data_bank 表

```
generate
for (i = 0; i < 4; i = i + 1)begin: data_way0
    data_bank_ram data_way0(
        .clka (clk),
        .wea (data_way0_wen[i]),
        .addra (data_way0_index[i]),
        .dina (data_way0_wdata),
        .douta (way0_data[i])
    );
end
endgenerate
```

一个 Cache 行有 16 个字节,分为 4 个 data_bank 表。

每一路的 data_bank 表为一个 RAM 256 * 32(深度 * 宽度)。

(b) tag,v 表

```
tagv_ram tagv_ram_way0 (
```

```

.clka (clk),
.wea (tagv_way0_wen),
.addra (tagv_way0_index),
.dina ({tagv_way0_wdata}),
.douta ({way0_tag,way0_v})
);

```

每个 Cache 行有一个 tag 和 v 域,表示该行是否有效和该行的在主存中的地址。

由于所有 Cache 操作对 tag 和 v 的操作是完全一致的,所以,tag 和 v 合并在一起存储,tag 占高位,v 占低位。

每一路的 tagv 表用一个 RAM 256 * 21(深度 * 宽度) 存储。

(c) d 表

```

d_regfile d_way0(
.clk (clk),
.resetn (resetn),
.addr (d_way0_index),
.wen (d_way0_wen),
.wdata (d_way0_wdata),
.rdata (way0_d)
);

```

每个 Cache 行有一个 d 域,表示该行的脏位,由于每行只有一个脏位,所以用 regfile 存储。

每一路的 d 表用 256 个 1 位的寄存器堆存储。

2. Cache 的内部控制逻辑设计

3. Cache 的控制包括两个状态机,主状态机和 write buffer 状态机。

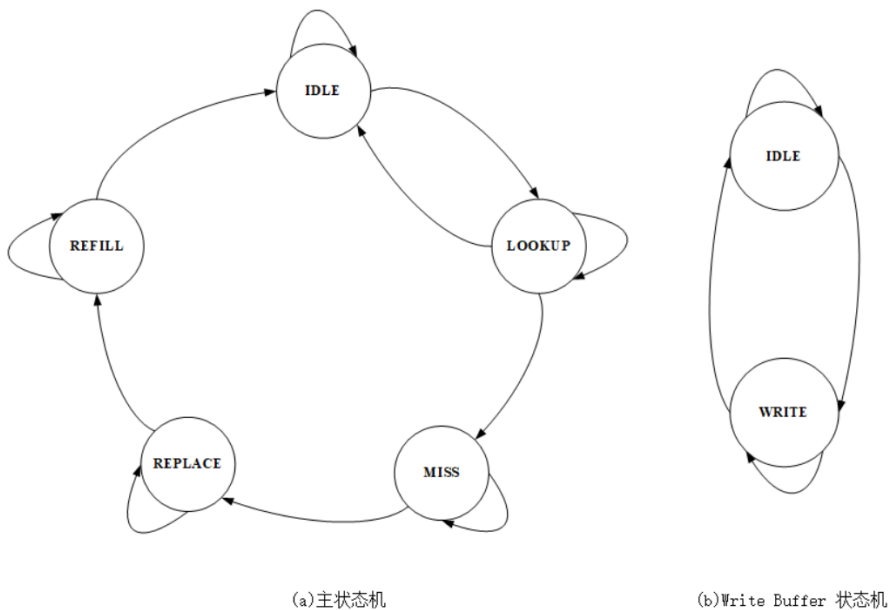


图 2: Cache 的状态机

主状态机包括 5 个状态,idle,lookup, miss, replace, refill。

(a) idle 状态

在 idle 状态下, Cache 等待外部的请求, 当有请求到来, 且与 Cache 中的 Hit write 不冲突时, 用组合逻辑拉高 `addr_ok` 信号, 并在下一拍 Cache 进入 lookup 状态。

```
assign addr_ok = main_current_state == LOOKUP & cache_hit & ~hit_write_conflict
               | main_current_state == IDLE & ~hit_write_conflict;
```

同时, 将请求信息中的 `index` 通过组合逻辑, 送到 Cache 进行查询, 将两路的对应的 Cache 行的数据 (`tag`, `v`, `data`) 在下一拍读出来。

并且, 在 request buffer 中存储请求的相关信息。

```
// request buffer, 存储接受到的请求信息
always @(posedge clk) begin
    if (~resetn) begin
        req_buffer_op <= 1'b0;
        req_buffer_index <= 8'b0;
        req_buffer_tag <= 20'b0;
        req_buffer_offset <= 4'b0;
        req_buffer_wstrb <= 4'b0;
        req_buffer_wdata <= 32'b0;
        req_buffer_type = 1'b0;
    end
    else if (addr_ok & valid) begin // next_state == LOOKUP, 存储请求信息
        req_buffer_op <= op;
        req_buffer_index <= index;
        req_buffer_tag <= tag;
        req_buffer_offset <= offset;
        req_buffer_wstrb <= wstrb;
        req_buffer_wdata <= wdata;
        req_buffer_type <= type;
    end
end
```

(b) lookup 状态

根据 Cache 的读出的两行的 `tag` 信息, 与锁存在 request buffer 中的请求信息进行比较, 如果有 Hit, Cache 进入 idle 状态, 写操作将数据传给 write buffer, 读操作直接返回 Cache 命中的数据。

否则进入 miss 状态。

```
// 请求的tag和Cache返回的tag比较, 判断是否命中
assign way0_hit = way0_v && (way0_tag == req_buffer_tag);
assign way1_hit = way1_v && (way1_tag == req_buffer_tag);
assign cache_hit = (way0_hit || way1_hit) && req_buffer_type;
```

为了在连续的 cache 命中时, 保证 `ipc` 为 1, 则如果在 lookup 命中, 且接受到流水线的有效的访存请求, 则直接在当前 lookup 状态下, 向 cache 的发起对下一个请求的查询, 将对应的 Cache 行的数据 (`tag`, `v`, `data`) 在下一拍读出来, 并在下一拍直接进入 lookup 状态, 进行比较。


```

if(ret_valid & ret_last)
    main_next_state = IDLE;
else
    main_next_state = REFILL;
endcase
end

```

如果是读请求,等到返回 cpu 请求的数据时(即 `miss_buffer_cnt == req_buffer_offset[3:2]`),即可拉高 `data_ok` 信号,将数据传给 cpu。不必等到 Cache 进入 idle 状态。

```

// 读请求, 等到返回cpu请求的数据时, 即可向cpu返回data_ok信号, 并将数据传给cpu
assign data_ok = (main_current_state == REFILL & ret_valid
    & miss_buffer_cnt == req_buffer_offset[3:2] & ~req_buffer_op)
    | ...;

```

• 在 CPU 中集成 ICache。

以 tlb 地址映射为例,如下图所示:

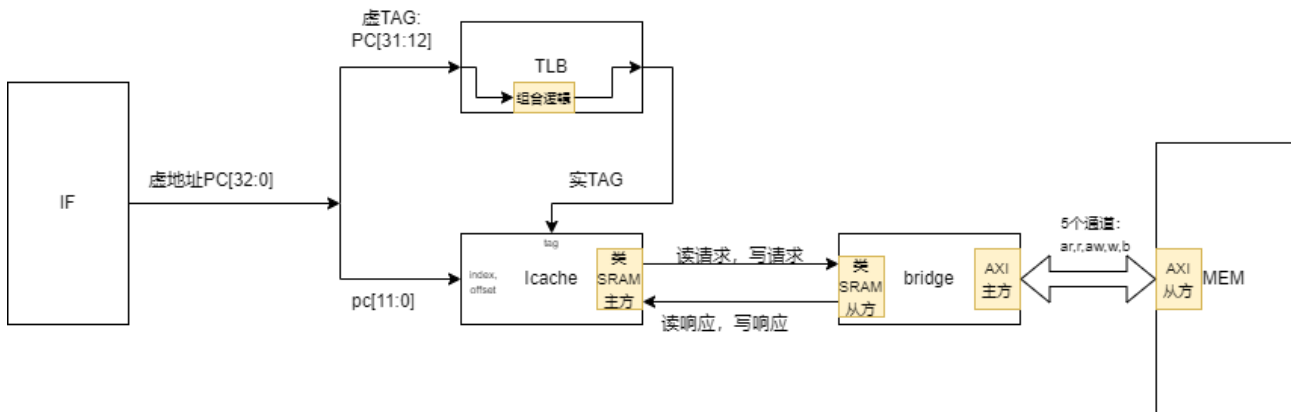


图 4: ICache 集成在 CPU 中示意图

1. ICache 与 CPU 的接口设计

由于 Cache 采用的是“虚 Index 实 Tag”的方式,所以当 cpu 取指的时候,把虚拟地址的 Index 直接传递给 Cache,Cache 便开始根据虚拟地址的 Index 进行查询,

同时,将虚拟地址传递给虚实地址转换单元,它能够通过组合逻辑在同一拍,将虚拟地址转换为物理地址的 TAG,并传递给 Cache。

上述方式使得 Cache 和 TLB 查询能并行执行,提高了效率。

2. ICache 与转接桥的接口设计

ICache 将访存请求发送到转接桥,转接桥将类 sram 的访存请求转换为 AXI 总线的访存请求,并且如果是多个字的访存请求,采用突发 burst 的方式。

以写通道的 burst 控制为例,

```

assign awlen = awtype_reg == 3'b100 ? 8'd3 : 8'd0; // 如果是写整个cache行, burst长度为3
assign awsize = 3'b010; // 每拍发送一个字
assign awburst = 2'b01; // 地址增加突发
assign wlast = awtype_reg == 3'b100 ? wdata_cnt == 2'b11 // 最后一个字, 拉高wlast

```

```

:1'b1;

always @(posedge clk)begin
    if(~resetn)begin
        wdata_cnt <= 2'b0;
    end
    else if(aw_current_state == AW_SEND_DATA && wready)begin
        wdata_cnt <= wdata_cnt + 1;
    end
end
end

```

• 在 CPU 中集成 DCache。

与 ICache 集成在 CPU 中的结构类似,区别主要在于 DCache 与 bridge 之间的写请求和写响应是由意义的,而在 ICache 中因为不会有写操作因此 icache 模块写请求和写响应相关的信号无意义。

1. DCache 与 CPU 的接口设计

Dcache 的接口接法与 Icache 类似,输入信号主要来自于 EX 模块,输出信号中的地址 OK 信号发送给 EX 模块,数据 OK 信号发送给 MEM 模块。Dcache 同样采用“虚 Index 实 Tag”的方式。

2. DCache 与转接桥的接口设计

与 ICache 相比,与转接桥相连的写请求与写响应端口不能再无视了。

从 DCache 中连接至转接桥的 wr_req, wr_type, wr_addr, wr_data, wr_wstrb 均在 cache 模块中赋值后发送。

```

assign wr_req = first_clk_of_replace & (req_buffer_type ? (replace_d & replace_v)
: req_buffer_op); // non-cache write
assign wr_data = req_buffer_type ? replace_data : {4{req_buffer_wdata}};
assign wr_addr = req_buffer_type ? {replace_tag, req_buffer_index, 4'b0}
: {req_buffer_tag, req_buffer_index, req_buffer_offset};
assign wr_type = req_buffer_type ? 3'b100 : 3'b010;
assign wr_wstrb = req_buffer_type ? 4'b1111 : req_buffer_wstrb;

```

传给 DCache 的 wr_rdy 和 wr_bvalid 信号分别连接至转接桥的 data_sram_wr_addr_ok 和 data_sram_wr_data_ok。

```

assign data_sram_wr_addr_ok = aw_current_state == AW_WAIT;
assign data_sram_wr_data_ok = b_current_state == B_WAIT;

```

3. 非缓存

在 cache 模块中添加一个一位宽的表示是否可缓存的变量 type。

icache 和 dcache 的 type 的值均在 csrfile.v 内做判断。直接地址翻译模式下存储类型由 CRMD 的 DATF 和 DATM 域决定,直接映射地址翻译模式下由命中窗口的 MAT 域决定,页表映射地址翻译模式下由所用页表项的 MAT 域决定。

```

assign inst_type = (~csr_crmd_pg) ? (csr_crmd_datf==2'b01) :
    hit_dmw0 ? (csr_dmw0_mat==2'b01) :
    hit_dmw1 ? (csr_dmw1_mat==2'b01) :
    s0_mat;

```

```

assign data_type = (~csr_crmd_pg) ? (csr_crmd_datm==2'b01) :
    hit_dmwo ? (csr_dmwo_mat==2'b01) :
    hit_dmwl ? (csr_dmwl_mat==2'b01) :
    s1_mat;

```

之后再根据传进 cache 模块的 type 变量对 cache 模块做修改。确保发生非缓存时不对 cache 缓存做任何修改。

以非缓存的 load 指令为例，确保 LOOKUP 状态不看比较结果直接进入 MISS 状态，且不向 Cache 发送读操作，也不产生对外的替换写。

```

assign cache_hit = (way0_hit || way1_hit) && req_buffer_type;
assign data_ok = (main_current_state == REFILL & ret_valid & (((miss_buffer_cnt ==
    req_buffer_offset[3:2]) & req_buffer_type) | ~req_buffer_type) & ~req_buffer_op) //
    read miss
    |(main_current_state == LOOKUP & (cache_hit | req_buffer_op)); // hit or
    write
assign data_way0_wen[i] = (wr_current_state == WR_WRITE & w_buffer_way == 0 &
    w_buffer_bank == i)? w_buffer_wstrb
    :(main_current_state == REFILL & replace_way ==0 &
    ret_valid & miss_buffer_cnt == i & req_buffer_type) ?
    4'b1111
    :4'b0000;

```

• cacop 指令的添加。

1. CACOP 指令基本数据通路

首先需要添加 cacop 指令的译码信号，然后在流水线中搭建数据通路

```

wire inst_cacop;
assign inst_cacop = op_31_26_d[6'h01] & op_25_22_d[4'h8];
assign cacop_code = inst_ID[4:0]; //cacop指令码传入两个Cache
assign icacop = inst_cacop & cacop_code[2:0] == 3'b000; //该信号传入ICache
assign dcacop = ex_inst_cacop & ex_cacop_code[2:0] == 3'b001; //该信号传入DCache

```

2. Cache 模块中对 CACOP 指令的实现

的 cacop 指令都可能涉及对于 tagv 的读写，以 way0 为例，当为初始化指令时，将指定 Cache 行的 tagv 写为 0；当为一致化指令时，将指定 Cache 行的 tag 保留，将 valid 位置为 0。详细代码如下：

```

// tag, v table
assign tagv_way0_index = (main_current_state == LOOKUP || main_current_state == IDLE) ?
    index // look up
    : req_buffer_index; // replace and refill;
assign tagv_way0_wen = main_current_state == REFILL & replace_way == 0 & ret_valid &
    ret_last & req_buffer_type
    | cacop & offset[0] == 0 & code[4:3] != 2'b10
    | cacop_wr_tagv & way0_v & (way0_tag == tag) & req_buffer_type &
    code[4:3] == 2'b10;

```

```

assign tagv_way0_wdata = {req_buffer_tag, 1'b1} & {21{ret_valid & ret_last &
    replace_way == 0 & req_buffer_type}}
    | {reg_tagv_dcacop[20:1], 1'b0} & {21{cacop & offset[0] == 0 &
        (code[4:3] == 2'b01 | code[4:3] == 2'b10)}}
    | {21'b0} & {21{cacop & offset[0] == 0 & code[4:3] == 2'b00}};

```

此外, ICache 没有其他操作, 但 DCache 可能涉及到写内存操作, 此时需要一个标志信号来记录本指令是否需要写内存, 代码如下:

```

wire cache_write_new;
assign cache_write_new = (offset[0] ? way1_d : way0_d) & (offset[0] ? {way1_tag, way1_v}
    : {way0_tag, way0_v}) & cacop & code[4:3] == 2'b01
    | way0_d & way0_v & cacop & code[4:3] == 2'b10 & way0_hit
    | way1_d & way1_v & cacop & code[4:3] == 2'b10 & way1_hit;

reg cache_write_reg;
always @(posedge clk) begin
    if (!resetn) begin
        cache_write_reg <= 1'b0;
    end
    else if (cache_write_new) begin
        cache_write_reg <= 1'b1;
    end
    else if (~cacop) begin
        cache_write_reg <= 1'b0;
    end
end

reg cacop_hit;
always @(posedge clk) begin
    if (!resetn) begin
        cacop_hit <= 1'b0;
    end
    else if (cacop & cache_write) begin
        cacop_hit <= 1'b1;
    end
    else if (~cacop) begin
        cacop_hit <= 1'b0;
    end
end

assign cache_write = (cache_write_reg | cache_write_new) & cacop & (code[4:3] == 2'b01
    | code[4:3] == 2'b10 & (cacop_hit | cache_hit));

```

使用该信号控制状态机, 使其进入 MISS 状态

```

LOOKUP:
if(cache_hit & (~valid | ~addr_ok) & ~cache_write)
    main_next_state = IDLE;
else if(~cache_hit & ~cache_write)

```



```

        main_next_state = MISS;
    else
        main_next_state = LOOKUP;

MISS:
    if(~wr_rdy)
        main_next_state = MISS;
    else
        main_next_state = REPLACE;

```

在 MISS 状态下,其向 AXI 总线桥发出相应的访存信号,代码如下:

```

assign wr_req = first_clk_of_replace & (req_buffer_type ? (replace_d & replace_v)
: (cacop & code[4:3] == 2'b01 & (req_buffer_offset[0] ? way1_d:way0_d) &
    reg_tagv_dcacop[0] |cacop & code[4:3] == 2'b10 & cache_write) ? 1
: req_buffer_op); // non-cache write
assign wr_data = req_buffer_type | cacop ? replace_data_final : {4{req_buffer_wdata}};
assign wr_addr = req_buffer_type & ~cacop ? {replace_tag, req_buffer_index, 4'b0}
: cacop & code[4:3]==2'b01 ?
    {reg_tagv_dcacop[20:1],req_buffer_index,req_buffer_offset[3:1],1'b0}
: {req_buffer_tag, req_buffer_index, req_buffer_offset};

assign wr_type = req_buffer_type | cacop ? 3'b100 : 3'b010;
assign wr_wstrb = req_buffer_type | cacop ? 4'b1111 : req_buffer_wstrb;

```

其中,reg_tagv_dcacop 是用来寄存即将写入 dcache 中 tagv 的寄存器

```

reg [20:0] reg_tagv_dcacop;
always @(posedge clk) begin
    if (!resetn) begin
        reg_tagv_dcacop <= 20'b0;
    end
    else if (~(|reg_tagv_dcacop) & cacop & code[4:3] == 2'b01) begin
        reg_tagv_dcacop <= offset[0] ? {way1_tag,way1_v} : {way0_tag,way0_v};
    end
    else if (~(|reg_tagv_dcacop) & cacop & code[4:3] == 2'b10 & way0_hit) begin
        reg_tagv_dcacop <= {way0_tag,way0_v};
    end
    else if (~(|reg_tagv_dcacop) & cacop & code[4:3] == 2'b10 & way1_hit) begin
        reg_tagv_dcacop <= {way1_tag,way1_v};
    end
    else if((|reg_tagv_dcacop) & ~cacop) begin
        reg_tagv_dcacop <= 20'b0;
    end
end
end

```

3. CPU 中 ICache 对 CACOP 指令的实现

将 ICache 的 cacop 执行位于 ID 阶段,下一条指令取值之前。ICache 的 cacop 指令信号需要挤占 ICache

取指信号的通路,然后在 if 模块添加新的通路,代码如下:

```
assign inst_vindex = inst_sram_addr[11: 4] & {8{~icacop | cacop_code[4:3]==2'b10 |
    icacop_complete}}
    | icacop_addr[11: 4] & {8{icacop & ~icacop_complete &
    cacop_code[4:3]!=2'b10}};

assign inst_voffset =inst_sram_addr[ 3: 0] & {4{~icacop | cacop_code[4:3]==2'b10 |
    icacop_complete}}
    | icacop_addr[ 3: 0] & {4{icacop & ~icacop_complete &
    cacop_code[4:3]!=2'b10}};

assign pre_pc_pa =      (!csr_crmd_pg | inst_cacop & cacop_code[4:3] == 2'b00
    | inst_cacop & cacop_code[4:3] == 2'b01) ? pre_pc // direct translate
:pre_pc_map;          // enable mapping

assign hit_dmw0 =      csr_dmw0_plv_met & csr_dmw0_vseg == pre_pc[31:29];
assign hit_dmw1 =      csr_dmw1_plv_met & csr_dmw1_vseg == pre_pc[31:29];

assign pre_pc_map =      hit_dmw0 ? {csr_dmw0_pseg, pre_pc[28:0]} // direct map
    windows 0
:hit_dmw1? {csr_dmw1_pseg, pre_pc[28:0]} // direct map windows 1
:(s0_ps == 6'b010101) ? {s0_ppn[19:9], icacop_vaddr[20:0]} // tlb map: ps 4Mb
:{s0_ppn,icacop_vaddr[11:0]};          // tlb map : ps 4kb
```

其中的 icacop_complete 信号,用来标记 ICache 的 cacop 指令是否执行完毕.

```
always @(posedge clk) begin
if (~resetn) begin
    icacop_complete <= 1'b1;
end
else if (icacop & cacop_code[4:3] != 2'b10) begin
    icacop_complete <= 1'b1;
end
else if (icacop_next & cacop_code[4:3] == 2'b10) begin
    icacop_complete <= 1'b1;
end
else
    icacop_complete <= 1'b0;
end
```

当 op 为 10 时,需要将 cacop 指令的虚拟地址 s0_vppn, s0_va_bit12 传入 tlb,代码如下:

```
assign icacop_vaddr = pre_pc & {32{~icacop| cacop_code[4:3]!=2'b10|icacop_complete}}
    | icacop_addr&{32{icacop & ~icacop_complete &cacop_code[4:3]==2'b10}};
assign {s0_vppn, s0_va_bit12} = icacop_vaddr[31:12]; // output to tlb
```

4. CPU 中 DCache 对 CACOP 指令的实现

dcache 的主要操作与 icache 相类似,此处不再赘述。

```

assign sram_addr_pa = (!csr_crmd_pg | ex_cacop & ex_cacop_code == 5'b00001 | ex_cacop
& ex_cacop_code == 5'b01001) ? ex_alu_result // direct translate
: sram_addr_map; // enable mapping

assign hit_dmw0 = csr_dmw0_plv_met & csr_dmw0_vseg == ex_alu_result[31:29] &
~(dcacop & ex_cacop_code[4:3] != 2'b10);
assign hit_dmw1 = csr_dmw1_plv_met & csr_dmw1_vseg == ex_alu_result[31:29] &
!hit_dmw0 & ~(dcacop & ex_cacop_code[4:3] != 2'b10);

assign sram_addr_map = hit_dmw0 ? {csr_dmw0_pseg, ex_alu_result[28:0]} // direct map
windows 0
:hit_dmw1? {csr_dmw1_pseg, ex_alu_result[28:0]} // direct map windows 1
:(s1_ps == 6'b010101) ? {s1_ppn[19:9], ex_alu_result[20:0]} // tlb map: ps 4Mb
:{s1_ppn,ex_alu_result[11:0]}; // tlb map : ps 4kb
//
wire dcacop;
assign dcacop = ex_cacop & (ex_cacop_code[2:0] == 3'b001);

```

二、 实验过程中遇到的问题、对问题的思考过程及解决方法(比如 RTL 代码中出现的逻辑 bug, 逻辑仿真和 FPGA 调试过程中的难点等)

• Cache 模块向 cpu 的 data_ok 信号设置。最初, 由于下意识认为 cpu 是串行工作, 所以不论是读还是写, 都等到 Cache 的状态机走完一圈后, 返回 idle 状态, 再向 cpu 返回 data_ok 信号。

但是, 实际上, 流水线和 Cache 是并行工作的, 所以只要 Cache 可以返回读数据或者接受了写数据, 就可以向 cpu 返回 data_ok 信号, 让流水线可以继续工作。

至于要保证 Cache 采用阻塞式设计(即一旦发生缺失, Cache 就不再接受新的请求, 直到缺失的数据返回), 需要用 addr_ok 信号来控制, 等到 Cache 的状态机走完一圈后, 接受完缺失的数据, 再接受新的请求。

```

assign addr_ok = main_current_state == LOOKUP & cache_hit & ~hit_write_conflict //
// 如果发生数据缺失, Cache拉低addr_ok, 不接受新的请求
|main_current_state == IDLE & ~hit_write_conflict;
assign data_ok =
(main_current_state == REFILL & ret_valid & miss_buffer_cnt == req_buffer_offset[3:2] &
~req_buffer_op)
// 读缺失时, 在refill状态下, 等到返回cpu请求的数据时, 即可向cpu返回data_ok信号
|(main_current_state == LOOKUP & (cache_hit | req_buffer_op));
// 写操作, 不管是否命中, 都将向cpu返回data_ok信号, 表示请求已经接受

```

三、 小组成员分工合作情况

王敬华负责 cacop 指令的实现。

李霄宇负责集成 dCache 到 cpu 中, 非缓存的添加。

艾华春负责 Cache 模块设计, 集成 iCache 到 cpu 中。

实验报告为根据每人负责代码的部分, 写相应部分的报告。